Specifying and Monitoring Non-functional Properties

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

M.Sc. Slim Kallel
aus Sfax, Tunesien

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferenten:	Prof. Dr.-Ing. Mohamed Jmaiel
	Prof. Dr.-Ing. Mario Südholt

Tag der Einreichung:	08. Dezember 2010
Tag der mündlichen Prüfung:	11. Juli 2011

Erscheinungsjahr 2011

Darmstadt D17

To my parents
To my wife and my daughter
To my brother, my sisters, and their children

Acknowledgments

First and foremost I offer my sincerest gratitude to my supervisors, Prof. Dr. Mira Mezini (Software Technology Group, TU Darmstadt, Germany) and Prof. Dr. Mohamed Jmaiel (ReDCAD Laboratory, University of Sfax, Tunisia).

I would have never finished this thesis without the help and the support of Mira Mezini. She always provides me with continuous advice and guides me to further clear up my ideas. I appreciate her patience during the first meetings and her support to continue working and to improve contributions and results. I am especially grateful to her for accepting me in her group as a DAAD Scholarship holder and for giving me the opportunity to obtain the German PhD diploma.

Mohamed Jmaiel is not just my supervisor but my career idol. Really, I dream to be like him. I started working with him since 2004 at the end of my engineer's studies, and after that in my master project. He always consistently encourages me and guides me to improve my work. His patience, support, and knowledge have been the key to my becoming what I am today.

Additionally, I would like to thank Dr. Anis Charfi for his enormous help to overcome many crisis situations especially in the first days of my stay in Darmstadt. He always contributed to improve my PhD work and he helped me also to further improve my writing of scientific papers.

I also would like to thank Prof. Dr. Mario Südholt for accepting to be the second advisor of this PhD work.

Moreover, I express my gratitude to the German Academic Exchange Service (DAAD) for supporting my PhD studies at Darmstadt University of Technology financially. Great appreciation goes to Ms. Cornelia Hanzlik-Rudolph, the contact person for DAAD scholarship holders from north Africa.

Many thanks go to all my colleagues at the Software Technology Group. I am grateful to Eric Bodden for proofreading this thesis and providing valuable comments. I thank also all my colleagues at the ReDCAD laboratory, especially Riadh Ben Halima, Bechir

Zalila, Sihem Loukil, and Rim Tayeb. Moreover, I thank Prof. Dr. Ahmed Hadj Kacem, Dr. Mourid Marrakchi and Ahmed Ben Arab for their support and their help during the last two years.

Special thank go to my Egyptian friends Mohamed Saied Emam and Wael Said Tawah and their families for their help and their support during my stay in Darmstadt. I will never forget them.

Last but certainly not least, I thank my family for their backup. I owe much of my success to my parents. I am indebted to my wife Fatma Krichen for her support during the difficult time of this thesis. I thank also my sisters and my brother for their love.

Abstract

This thesis focuses on the implementation and the control of non-functional safety properties during system execution. More concretely, it describes the development process of such properties, starting with the formal specification, the verification, and the runtime enforcement of the specified properties to avoid any undesired behavior.

This thesis starts by studying and classifying the approaches on the specification and runtime verification of non-functional safety properties. When examining these approaches, the following observations are made. First, the non-functional properties are generally ignored in the early phases of the software development process. They are often addressed after the functional part is implemented, which has negative effects on the quality of the code. The approaches that use UML for modeling these properties cannot verify the absence of contradictions between the specified properties. In addition, UML lacks means to express various types of non-functional properties, such as temporal properties. Second, runtime verification approaches monitor the execution of the application at runtime and detect violations of the specified properties. However, just detecting the violation is not sufficient for critical applications. These approaches should enforce these properties and avoid the misbehavior of the system by skipping the execution of undesired events. Third, in current approaches the code for enforcing non-functional properties is mostly not encapsulated in separated modules. The implementation cuts across the functional application code. This lack of modularity leads to serious problems related to the quality of code and the possibility of changing those properties.

The thesis shows a generic and holistic approach, called Seven-pro that combines formal methods and aspect-oriented programming for specifying and runtime enforcing non-functional safety properties. Seven-pro covers the whole development process of non-functional properties and avoids the gap between the specification and the implementation by automatically generating aspects from a high-level specification. The generated aspects will be integrated, in modular way, in the functional application code for enforcing the formally specified properties at runtime.

In addition, this thesis shows how Seven-pro covers different types of non-functional

properties in distributed applications. This approach is applied to structural, qualitative and quantitative behavioral non-functional properties. This thesis presents three applications for the supported types of properties.

In the context of structural properties, Seven-pro is applied for specifying and enforcing architectural properties of distributed object-oriented applications that are characterized by dynamic software architectures. Seven-pro uses a combination of Z notation and Petri nets to specify (a) the architectural styles with their architectural invariants, (b) the reconfiguration operations with their pre- and post-conditions, and (c) the coordination protocols describing the execution order of the reconfiguration operations. A verification step is performed to verify the consistency of the specification and the preservation of the architectural style after a reconfiguration of the architecture. After that, the Z and Petri nets specifications are automatically translated to AspectJ aspects to verify - before each reconfiguration operation - that all related architectural properties are satisfied.

In the context of qualitative behavioral properties, Seven-pro is applied for specifying and enforcing static and dynamic separation of duties and different types and characteristics of delegation policies on top of role-based access control. In the specification phase, TemporalZ, a combination of Z notation and linear temporal logic, is used for formally specifying the supported policies. In the verification phase, the absence of contradictions between the specified policies is verified. In the implementation phase, the aspect language Alpha is extended with a new library for supporting the specified properties. In addition, TemporalZ specifications are automatically translated to Alpha aspects to control the access permissions according to the specified policies.

In the context of quantitative behavioral properties, Seven-pro is applied for specifying and enforcing temporal properties in Web service compositions. To support both relative and absolute timed properties, a new formal language called XTUS-Automata is proposed which extends timed automata with the constructs of the XTUS language. After formally verifying the absence of deadlocks in timed automata specifications and verifying other properties related to the XTUS language, the XTUS-Automata specifications are automatically translated to AO4BPEL aspects.

Zusammenfassung

Schwerpunkt dieser Dissertation ist die Implementierung und Durchsetzung von nicht-funktionalen Sicherheitseigenschaften (safety properties) während der Programmausführung. Konkret wird ein Entwicklungsprozess beschrieben, der sicherstellt, dass die Software die gewünschten Eigenschaften aufweist. Dieser reicht von der formalen Spezifikation über die Verifikation bis zur Durchsetzung der spezifizierten Eigenschaften zur Laufzeit. Dies ermöglicht es unerwünschtes Verhalten zu vermeiden.

Diese Dissertation beginnt mit einer Studie und Klassifikation der Ansätze zur Spezifikation und Laufzeitverifikation von nicht-funktionale Sicherheitseigenschaften. Dabei wurden folgende Beobachtungen gemacht: Erstens werden nicht-funktionalen Eigenschaften während der frühen Phasen des Softwareentwicklungsprozesses weitestgehend ignoriert; sie werden meist erst berücksichtigt, wenn der funktionale Teil bereits implementiert ist was sich negativ auf die Codequalität auswirkt. Die Ansätze, die UML zur Modellierung dieser Eigenschaften benutzen, können die Widerspruchsfreiheit bezüglich der spezifizierten Eigenschaften nicht beweisen. Des Weiteren fehlt es der UML an Möglichkeiten verschiedene Arten nicht-funktionaler Eigenschaften, wie z.B. zeitbezogene Eigenschaften, auszudrücken. Zweitens, Ansätze, die das Verhalten der Anwendung zur Laufzeit beobachten und verifizieren und dadurch Verletzungen detektieren, sind im Falle kritischer Anwendungen nicht ausreichend. Solche Ansätze sollten die geforderten Eigenschaften durchsetzen und ein Systemfehlverhalten durch Unterdrücken der Ausführung unerwünschter Ereignisse vermeiden. Drittens, der Code zur Durchsetzung der nicht-funktionalen Anforderungen ist in aktuellen Ansätzen nicht gut modularisiert. Die Implementierung durchschneidet vielmehr den funktionalen Kern der Anwendung. Dieser Mangel an Modularität führt zu schwerwiegenden Problemen bezüglich der Codequalität und der Änderbarkeit der Sicherheitsanforderungen.

Diese Dissertation verfolgt einen generischen, holistischen Ansatz namens Seven-pro, der formale Methoden und aspekt-orientierte Programmierung kombiniert, um nicht-funktionale Sicherheitsanforderungen sowohl zu spezifizieren als auch durchzusetzen. Seven-pro deckt den gesamten Entwicklungsprozess nicht-funktionaler Anforderungen ab und vermeidet

Diskrepanzen zwischen Spezifikation und Implementierung durch das automatische Generieren von Aspekten basierend auf der Spezifikation. Die generierten Aspekte werden auf modulare Art und Weise mit dem funktionalen Kern der Anwendung integriert, um die formal spezifizierten Eigenschaften zur Laufzeit durchzusetzen.

Zusätzlich zeigt diese Dissertation, wie Seven-pro verschiedene Arten von funktionalen Anforderungen im Kontext verteilter Anwendungen abdeckt. Der Ansatz wird dabei auf strukturelle, qualitative und quantitative nicht-funktionale Eigenschaften des Verhaltens angewendet. Die Arbeit präsentiert drei Anwendungen für die unterstützten Arten von Eigenschaften.

Im Kontext struktureller Eigenschaften wird Seven-pro zur Spezifikation und Durchsetzung architektonischer Eigenschaften von verteilten, objekt-orientierten Anwendungen angewandt welche eine dynamische Softwarearchitektur aufweisenen. Seven-pro verwendet eine Kombination aus Z-Notation und Petrinetzen zur Spezifikation (a) des Architekturstils und von Architekturinvarianten, (b) der Rekonfigurationsoptionen mit Vor- und Nachbedingung, sowie (c) der Protokolle zur Koordination der Reihenfolge der Rekonfigurationsoperationen. Ein Verifikationsschritt wird durchgeführt, um die Konsistenz der Spezifikation und den Erhalt des Architekturstils nach deren Rekonfiguration sicherzustellen. Daran anschließend werden die Z- und Petrinetzspezifikationen automatisch in AspectJ-Aspekte übersetzt, um vor jeder Rekonfigurationsoperation sicherzustellen, das alle zugehörigen Architektureigenschaften erfüllt sind.

Im Kontext qualitativer Verhaltenseigenschaften wird Seven-pro eingesetzt zur Spezifikation und Durchsetzung statischer und dynamischer Aufgabentrennungs- und Delegationsrichtlinien auf Basis von rollenbasierten Zugangsbeschränkungen. Während der Spezifikationsphase wird TemporalZ, eine Kombination aus Z-Notation und linearer Temporallogik benutzt um die unterstützten Richtlinien formal zu beschreiben. Während der Verifikationsphase werden die spezifizierten Richtlinien auf Widerspruchsfreiheit geprüft. In der Implementierungsphase wird die Aspektsprache Alpha mit einer neuen Bibliothek, die die spezifizierten Eigenschaften unterstützt, erweitert. Des weiteren werden TemporalZ-Spezifikationen automatisch in Alpha-Aspekte übersetzt, die Zugriffsrechte gemäß der spezifizierten Richtlinien beschränken.

Im Kontext quantitativer Verhaltenseigenschaften wird Seven-pro zur Spezifikation und Durchsetzung zeitbezogener Eigenschaften einer Komposition von Web Services eingesetzt. Um sowohl relative als auch absolute zeitbezogene Eigenschaften zu unterstützen, wird eine neue formale Sprache mit dem Namen XTUS-Automata vorgeschlagen, die zeitbezogene Automaten um die Konstrukte der XTUS-Sprache erweitert. Nachdem sowohl die Deadlock-Freiheit als auch andere, XTUS-bezogene Eigenschaften bewiesen worden sind, werden die XTUS-Automata-Spezifikationen automatisch in AO4BPEL-Aspekte übersetzt.

Contents

List of Figures

List of Listings

Chapter 1
Introduction

1.1 The Thesis in a Nutshell

Software systems are usually characterized by functional and non-functional properties. While functional properties specify the structure and the behavior of the system, non-functional properties address software qualities like maintainability, extensibility, reusability, and system efficiency like reliability, availability, security, etc. Both kinds of properties are relevant to software development. However, in practice, based on different proposed methodologies and languages, the designer and the developer often focus on functional properties and do not directly address the non-functional ones. The latter are generally ignored in the early phases of the software development process and managed in an ad-hoc manner without following a rigorous process.

There are two classes of non-functional properties. The first class corresponds to the properties that can be monitored and verified at runtime, like security properties, time related constraints, architectural invariants, etc. The second class corresponds to the properties that represent qualities of software such as portability, reusability, maintainability, adaptivity, etc. Properties of the first class are called non-functional safety properties. The main use of non-functional safety properties is for monitoring and for verifying at runtime the execution of software systems. The runtime verification mechanisms are limited to monitor the execution of systems and verify if their properties are satisfied. Enforcement mechanisms, on the other hand, additionally ensure that bad things such as faults and deadlocks do not happen. They allow controlling the execution and prohibiting (i.e., skipping) any critical operation that do not satisfy a safety property.

There are two main phases for the runtime enforcement of non-functional properties. The first phase consists in the specification of these properties. Formal methods are well suited for specifying such properties and for reducing their complexity. Formal verification mechanisms ensure that the specification of these properties is consistent and does not contain any contradiction. The second phase consists of using an enforcement mechanism

for implementing the formally specified properties. The enforcement code can be written manually or generated automatically from the high-level specification.

Unfortunately, the code for enforcing non-functional properties is often scattered throughout the functional code of the application and also tangled with code related to other properties. These properties are mostly changed according to the user requirements or the execution environment. As a result, having a modular implementation of non-functional properties becomes necessary to master the complexity and the frequent change of these properties.

There is also a gap between the specification and the implementation phase of the non-functional properties. Generally, the code that checks and enforces these properties is written manually. Besides being a tedious activity, such a manual translation has the more severe problem that there is no guarantee that the result actually conforms to the specification: The code that implements the properties may contain contradictions that did not exist in the specification. An automatic and correct code generation can overcome the previous limitations.

The goal of this PhD work is to define a generic approach for implementing non-functional safety properties. This approach combines formal methods and Aspect-Oriented Programming (AOP). Formal methods are used for formally specifying different types of non-functional properties. Theorem provers and model checkers can be used for ensuring the consistency of the specification and proving other application-specific properties. Aspect-oriented programming, as the most appropriate solution for solving crosscutting modularity problems, is used for enforcing the formally specified properties at runtime. The formal specification of non-functional safety properties is automatically translated to aspect code.

To illustrate the proposed approach, this thesis presents three case studies for specifying and enforcing non-functional safety properties. These case studies address different types of properties and different application domains. The considered properties are: architectural properties, behavioral qualitative properties, and behavioral quantitative properties.

When dealing with architectural properties, we focus on the specification and the enforcement of architectural invariants and coordination protocols of dynamic software architectures for object-oriented applications. For behavioral qualitative properties, this thesis shows how the proposed approach can be applied for specifying and enforcing separation-of-duties and delegation policies on top of the role-based access control policy. As an example of behavioral quantitative properties, this thesis describes the specification and the enforcement of absolute and relative temporal properties in Web service composition.

1.2 Contributions

The main contribution of this thesis is to provide a holistic and generic approach for implementing non-functional properties. This approach, called SEVEN-pro, combines formal methods and aspect-oriented programming for specifying and enforcing non-functional properties. It covers the main phases of the development process: formal specification, structural validation, formal verification, and runtime enforcement of the specified properties. We applied, our approach to three applications domains, namely, software architecture, access control protocols and service oriented applications. When addressing these application domains we dealt with structural and behavioral non-functional properties. To be able to formally specify and enforce the different kinds of properties, it was necessary to define domain specific languages, by combining existing ones, and to extend some aspect oriented languages. This led to the following contributions:

Application of the SEVEN-pro approach for specifying and enforcing architectural properties.

- **Combination of Z notation and Petri nets**
  For formally specifying static, dynamic and coordination aspects of dynamic software architectures of object-based applications, a combination of the Z notation and Petri nets *Z-PN* is proposed. The Z notation is used to formally specify the architectural styles, their architectural invariants, and the reconfiguration operations as well as their pre- and post-conditions. Petri nets are used to specify the coordination protocols that define the execution order of reconfiguration operations. The consistency and the preservation of the system after an architectural reconfiguration are formally verified using a theorem prover.

- **Automatic transformation of Z-PN to AspectJ**
  A holistic process is proposed for generating aspect code from a high level specification of architectural properties. The Z predicates and the specified Petri nets transitions are automatically translated into aspect code. This generation process is based on well-defined transformation rules and aspect templates. As a proofs-of-concept, a prototype of AspectJ generator *ZPN-2-AspectJ* is implemented.

Application of the SEVEN-pro approach for specifying and enforcing access-control policies.

- **A domain specific specification language for access control policies**
  Temporal$\mathcal{Z}$, a formal language that combines the Z notation and linear temporal logic, is extended by specific security predicates for specifying static and dynamic separation of duties and different types and characteristics of delegation on top of the RBAC model. Conflicts between RBAC constrains, and between separation-of-duties and delegation constraints are resolved.

- **A domain specific implementation language for access control policies**
  For encoding the specified policies at the implementation level, the aspect-oriented language ALPHA is used. This language is extended by a specific security library to easily translate the specified properties into ALPHA aspects. This library contains ALPHA queries which correspond to the implementation of the *TemporalZ* security predicates.

- **From a domain specific specification language to a domain specific implementation language**
  An automatic code-generation process is defined which translates *TemporalZ* specifications (i.e., as a domain-specific specification language) into ALPHA code (i.e., as a domain-specific implementation language). The generated aspects will be woven into the functional application code to enforce the specified properties. A research prototype of this generator is implemented.

Application of the SEVEN-pro approach for specifying and enforcing absolute and relative temporal properties in Web-service composition.

- **A formal language for absolute and relative temporal properties: XTUS-Automata**
  A new formal language *XTUS-Automata* is defined, which combines temporal automata and the XTUS language for supporting the specification of absolute and relative time related properties in Web service composition. In this language, an XTUS expression is written under the corresponding automata transition. To formally define this new language, the semantics of the XTUS language is specified using the Z notation. As proof-of-concept, a GUI is implemented to support the design of XTUS-Automata specifications.

- **An Approach for specifying temporal properties**
  Based on the XTUS-Automata language, an approach for modeling the composition of Web services and their temporal properties is proposed. Some specification patterns are also proposed for helping the designer to specify such properties.

- **AO4BPEL aspect generation process**
  A process for the automatic generation of AO4BPEL aspects from XTUS-Automata specifications is defined. Based on aspect templates, the pointcut is automatically generated for intercepting the corresponding BPEL activity and the advice skeleton is automatically completed by translating timed automaton guards and XTUS expressions into BPEL constraints. A prototype of the AO4BPEL aspect generator is implemented and integrated with the graphical user interface as an Eclipse plug-in.

1.3 Publications

The contributions and the obtained results of this thesis have been published in proceedings of conferences and journals as listed below:

1. Slim Kallel, Mohamed Hadj Kacem, and Mohamed Jmaiel. Modeling and Enforcing Invariants of Dynamic Software Architectures, *Software and Systems Modeling* (SoSyM), Springer, 2011. (to appear).

2. Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. A Holistic Approach for Access Control Policies: From Formal Specification to Aspect-based Enforcement, *International Journal of Information and Computer Security* (IJICS), Inderscience publishers. pages 337–354, Volume 3, Number 3/4, 2009.

3. Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. Specifying and Monitoring Temporal Properties in Web services Compositions, *In Proceedings of the 7th IEEE European Conference on Web Services* (ECOWS), IEEE Computer Society, pages 148–157, Eindhoven, The Netherland, November 2009.

4. Slim Kallel, Anis Charfi, Mira Mezini, Mohamed Jmaiel, and Karl Klose. From Formal Access Control Policies to Runtime Enforcement Aspects, *In Proceedings of the 1st International Symposium on Engineering Secure Software and Systems* (ESSoS), LNCS 5429, Springer, pages 16–31, Leuven, Belgium, February 2009.

5. Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. Aspect-based Enforcement of Formal Delegation Policies, *Proceedings of the 3rd International Conference on Risks and Security of Internet and Systems* (CRiSIS), IEEE, pages 9–17, Tozeur, Tunisia, October 2008.

6. Slim Kallel, Anis Charfi and Mohamed Jmaiel. Using Aspects for Enforcing Formal Architectural Invariants, *Electronic Notes in Theoretical Computer Science* (ENTCS), volume 215, Elsevier, pages 5–21, September 2008.

7. Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. Combining Formal Methods and Aspects for Specifying and Enforcing Architectural Invariants, *In Proceedings of the 9th International Conference on Coordination Models and Languages* (Coordination), LNCS 4467, Springer, pages 211–230, Paphos, Cyprus, June 2007.

1.4 Structure of the Thesis

This thesis consists of an introduction in Chapter 1, five chapters, and a conclusion.

- Chapter 2 presents the background information pertaining to this PhD work. This chapter also describes the approaches most related to the specification and runtime verification of non-functional properties.
- Chapter 3 describes the motivation of this work by discussing the limitations of existing approaches. After that, it introduces the SEVEN-pro approach, which we propose for specifying and enforcing non-functional properties. The last part of this chapter presents the validation of this approach using three cases studies considering different properties and different application domains.
- Chapter 4 presents the application of the SEVEN-pro approach for specifying and runtime-enforcing architectural properties: architectural invariants, pre- and the post-conditions, and coordination properties. This approach is applied to dynamic software architectures of object-oriented applications.
- Chapter 5 presents how to apply the SEVEN-pro approach to the role-based access control model. The separation of duties and delegation policies are formally specified on top of the role-based access control policy. These policies are also formally verified and automatically translated to aspect code.
- Chapter 6 presents the application of the SEVEN-pro approach for specifying and runtime-enforcing temporal properties in Web service compositions. Absolute and relative temporal properties are considered.
- Chapter 7 summarizes the results and contributions of this PhD work. Moreover, it presents a review of the proposed approach and the related tools. It also outlines several directions for future work.

Chapter 2
Background and Related Work

2.1 Introduction

This chapter is dedicated to the presentation of background knowledge and the related work about specifying and enforcing non-functional safety properties.

A short overview of different types of non-functional properties will be given. In addition, different mechanisms, methods and languages for formally specifying and runtime enforcing these properties will be also discussed and classified. Finally, Aspect-Oriented Programming as an enforcement mechanism used in our approach will be presented.

As related work, we will mention some interesting approaches focused on the specification of non-functional properties. We will present also different runtime verification approaches that automatically translate high-level specifications into corresponding code for runtime verifying the specified properties.

The remainder of this chapter is structured as follows. In Section 2.2, background knowledge about the relevant mechanisms and languages used in this PhD work is given. Sections 2.3 and 2.4 present the most related work on the specification and the runtime enforcement of non-functional properties, respectively. Section 2.5 concludes this chapter.

2.2 Background

This section provides the reader with a necessary background comprising brief definitions and reviews of the mechanisms and methods used in this PhD work. First, we classify non-functional properties and we define the class of properties that we are interested in. Second, we present an overview of the formal methods and the verification tools that can be used for specifying such properties. Third, we describe the automatic code generation associated with some industrial and research examples. Fourth, we present a classification of the enforcement mechanisms and finally, we detail Aspect-Oriented Programming as a powerful mechanism for runtime enforcement of non-functional safety properties.

2.2.1 Non-Functional Safety Properties

Distributed applications use Non-functional properties for different purposes. We classify these purposes into two classes:

First, these properties are used to provide a contract between the client and the service provider as a service-level agreement (SLA) in Web services based applications. The SLAs cover only the pre-defined properties (e.g., response time, execution time, etc.) which are related to the application as a whole. Theses SLAs do not present any properties related to the behavior of the individual components constituting the application as well as the relation between the components [200].

Second, non-functional properties are mainly used in runtime monitoring, which is a promising technique for verifying, achieving and maintaining a well-performing system. The user can verify at runtime if the specified properties are satisfied according to the system state. Despite that the properties can be formally specified and verified, they can be violated at runtime. To resolve this limitation, the runtime-monitoring code of non-functional properties can be generated automatically from a high level specification.

The non-functional properties are also classified into two categories:

- *Liveness properties* assert that "something good eventually happens during the program execution" [126], e.g., the accessibility and the guarantee of a service, etc.

- *Safety properties* assert that "nothing bad should happen during the program execution" [126]. For example, ensuring mutual exclusion, deadlock freedom, preserving invariants after a system reconfiguration, access control of resources, etc.

These two types of properties differ in how to ensure their *satisfaction*. If a safety property is violated, it is possible to notice it after a finite execution of the system. This violation can be detected, during the system execution, using runtime monitoring mechanisms [173] and can also be detected, in some cases, early with testing, simulation methods and model-checking tools. However, detecting violations of the liveness properties requires to *prevent* some infinite execution of the system.

In our work, we are interested only in non-functional safety properties. These properties ensure the proper functioning of software systems against undesirable behavior. These properties can express the structure and the architecture of the application, and can also be used to specify the behavior of the application, taking into account qualitative and quantitative aspects. In the remainder of this thesis, we use simply the term "non-functional properties" to describe such properties.

2.2.2 Formal Specification and Verification

Assuring the safety of complex systems aims to guarantee a certain level of confidence that there is no dangerous or unexpected behaviors. Ensuring the proper functioning of the system requires clear and rigorous control of the critical operations of the system and verification of the relevant properties. The formal methods are the most appropriate techniques to provide high assurance about the reliability, the dependability, and the safety of such systems.

Formal methods are based on mathematical foundations, enabling the specification and the development of systems, as well as the verification of their properties. A range of formal languages and techniques exist to treat different types of properties at different levels of the development process, but they are highly recommended especially in the early phases of the development process. They allow to specify the user requirements in a structured, precise, unambiguous, verifiable, and maintainable way. In addition, formal methods eliminate any ambiguous terminology or description and avoid the misunderstanding of the specification by users at all levels of the development process.

There is a wide range of possible criteria to classify formal methods. The classification based on the application area of the methods, proposed by Jeannette Wing [191] seems to be the most concrete. There is two broad classes called model- and property-oriented methods.

Model-oriented methods The specification of system consists in defining a model of the system in terms of mathematical structures such as relations, functions, set and sequences. There are two sub classes, depending on the type of the system to be specified. (i) Formal methods for sequential programs and abstract data types, such as: finite-state machines, VDM, B, and Z notation. (ii) Formal methods for concurrent and distributed programs, such as Petri net, Communicating Sequential processes (CSP), Calculus of Communicating Systems (CCS), I/O automata.

Property-oriented methods The specification of the system consists in defining some properties, usually in terms of axioms that should be satisfied by the system. There are two categories: (i) Axiomatic methods originate from the work of Hoare on proofs of correctness [93]. Iota and OBJ are formal languages that belong to this category. (ii) Algebraic methods: The processes are defined to be heterogenous algebras, such as temporal logic and LOTOS.

The use of one of these methods for specifying non-functional properties depends on the type of the properties and on the application domain. The most important advantage of using formal methods, compared of other design systems, is the use of formal verification

schemes. The verification tools are often used to guarantee the correctness and the consistency of the formally specified system. There are two techniques to prove the properties of a system.

Theorem proving This technique is used when the system is specified through mathematical definitions. Such system is verified using automatic/semi-automatic theorem provers, which are based on a library of axioms and a set of predefined inference rules. Theorem proving is independent of the size of the state space, and can therefore be applied to models with a large number of states, or even to models whose number of states is not determined.

Automated theorem proving allows to prove the properties of the system automatically, without human intervention. This technique is very expensive in terms of time and resources and is not practical for many complex specifications. Therefore, there exist interactive theorem provers which allow the designer to guide the proof. For example, Z/EVES [139] is a semi-automatic theorem prover, which allows to prove theorems for verifying specifications written in Z notation. After specifying the system using Z schemas and writing a Z theorem, the designer helps the prover by providing reduction commands (e.g., simplify, reduce, prove by reduce, prove by rewrite), commands for applying theorems, definitions and lemmata (e.g., apply, try, invoke, invoke predicate, Use), quantifiers (e.g., Instantiate, Prenex), etc.

Model checking This technique is used when the system is specified as a set of states connected by a set of transitions. Model checking verifies the satisfaction of a property by executing an algorithm that enumerates the list of possible reachable states that a system could enter during its execution. The effectiveness of model checking depends on the size of the space of reachable states. To improve the efficiency, the number of the accessible states is minimized using abstraction techniques generally guided by the designer.

To cover all aspects of a property, the designer is sometimes obliged to use two formal languages, which have two different semantics. There are two different alternatives. In the first one, the designer can use two languages separately. Each language is used to specify a separate aspect of a property. Two verification mechanisms can also be used to assure the required properties. For example, we separately used Z notation and Petri nets to specify respectively the structural aspect and the coordination aspect of the dynamic software architectures of distributed applications. The other alternative is to combine the two languages or extend one language by another. The combination should be formally defined to be supported by the existing verification tools. For example, we used a new formal language, called *Temporal*$\mathcal{Z}$, which combines Z notation and linear temporal logic. The temporal operators are formally defined in the framework of Z notation.

2.2.3 Automatic Code Generation

Automatic code generation is a useful technique to automatically construct code in a specific programming language from a high-level model. Code can be generated from:

- A set of requirements of the program written in a semi-formal model defined using UML diagrams and their extensions. Several commercial tools support the automatic generation of code from a UML model such as Omando under Eclipse [95] and IBM Rational Rose Developer for Java [94]. Other research approaches have been proposed for generating code from a UML profile of a specific application domain. For example, Basin et al. [23] generate code from secureUML, a profile for modeling access control policies. Vidal et al. [189] proposed an approach based on the MDA approach for generating code from MARTE profile (i.e. a UML extension to describe real-time properties).

- Formal specifications or mathematical models. In this case, the code generation is also known as program synthesis [22]. The translation of formal specifications to code has been addressed in several approaches such as Z and ObjectZ [97,165], B method [131], CSP-OZ [145], Petri nets [162], Automata [16], etc. For example, in [165], Ramkerthik and Zhang propose the generation of Java code with design contracts from an Object-Z Specification. This work translates the structure of Object-Z specifications to XML and then generates a Java skeleton by processing the XML representation. Another approach is proposed by Jia and al. [97] for synthesizing functional code from UML and Z. They translate UML models into a Z specification, which is used to generate C++ code.

Generally, the code-generation process consists of the following steps. The generator parses and analyzes the specification for checking syntax and type errors. After that, the specification is translated into code or code fragments. Finally, the generated fragments are assembled and integrated within the code of the application. Some powerful generators can optimize the generated code and even try to verify if this code satisfies the original specification. To facilitate the generation process, the developer can define well-defined transformation rules form a high level specification to code based also on templates of code.

The generation of code from a high level model has the potential to minimize errors, reduce the cost of the development in terms of time and efforts, and improve the quality of the code. The generated code is more likely correct than a handcrafted. However, the automatic code generator does not guarantee that the generated code satisfies the specification to be developed. In addition, if the model contains errors, these errors will be translated to code. For this reason and to minimize the limitations of code generation, we motivate the use of formal methods as techniques for specifying as well as verifying the model before generating code.

Several approaches have been proposed to automatically generate aspect code to profit from advances in Aspect-oriented programming [113]. These approaches are generally used to enforce or to verify non-functional properties at runtime. The generated aspect code can be woven in a modular way, without affecting the functional code of the original application. If the user requirements or the specifications change, new aspects can be deployed and replace the old ones. Using a runtime aspect weaver, the developer can modify/add new aspects that represent new requirements without stoping the deployed process. As examples of aspect generation approaches, Bodden et al. [32] propose to generate AspectJ aspects from LTL specifications. The approach presented in [88] allows also to generate AspectJ aspects from LSC specifications. Aloulou et al. [11] propose an approach to automatically generate JBoss AOP aspects from a formal specification of security policies written in Z notation.

2.2.4 Enforcement Mechanisms

Enforcement mechanisms allow to verify the application against a set of expected properties. The application should be executed without violating any properties. These mechanisms can be performed statically or dynamically. Hamlen et al. [87] classify the enforcement mechanisms into three classes: static analysis, program rewriting, and execution monitoring (i.e., runtime monitoring).

The *static analysis* [152] covers methods that analyze the code of the application before the execution of the program. The static analyzers detect in the application code errors that can be caught at runtime. The result of this analysis is the acceptance or the rejection of the controlled program: If the analyzer does not detect unsatisfied properties, the program is permitted to be executed unhindered. Otherwise, the program will be rejected until the programmer intervenes to modify the infected part of the code. For example, security policies and special control of information flow is an example of non-functional properties that can be enforced by static analysis [148].

The *program rewriting* is a powerful enforcement mechanism which allows to rewrite the controlled program in order to enforce the properties before the execution. The idea is to rewrite the part of the code that does not satisfy a property. The program rewriting tool takes the untrusted code as input and automatically transforms it to a new version satisfying the infected properties. During the execution, the new program should guarantee the same functionalities of the original program except those that do not satisfy the infected properties. [66, 67] propose approaches for enforcing different security policies based on program rewriting techniques for Java.

Execution monitoring monitors critical (i.e., sensitive) events of an application during execution. When one of these events is about to violate a safety property, an intervention will be performed immediately to terminate the execution of the undesirable event [173].

If all properties related to an event are satisfied, this event will be normally executed. Other powerful techniques have been proposed to extend this definition based on execution monitoring and modifying programs at runtime. Ligatti et al. [26,129] propose other types of intervention than the termination of the program. They can also suppress undesired or dangerous actions without necessarily terminating the program and also insert additional actions into the program at runtime.

To enforce non-functional safety properties, we should adapt an enforcement mechanism that supports the dynamic behavior of these properties: The satisfaction of these properties depends on the execution of the application. For this purpose, we dismiss the static analysis and the program rewriting mechanisms because they operate on the program before its execution. On the other hand, the execution monitoring mechanism controls the behavior of a program at runtime and intervenes when an imminent violation of a property is detected. This intervention consists of interrupting the execution of the undesired action that violates the property.

Based on the runtime enforcement monitoring, several approaches and languages have been proposed to enforce at runtime different types of non-functional safety properties. Section 2.4 presents the most important works in this field.

2.2.5 Aspect-Oriented Programming

Traditional programming languages make the evolution and the maintenance of large and complex applications relatively difficult. In addition to main functionalities, the code of these applications contains other important concerns such as logging, security, temporal properties, etc. which are generally dispersed and mixed. The code relative to these concerns is scattered throughout the functional code of the application and also tangled with code related to other concerns.

One of the solutions that have been proposed to resolve such problem is the separation of concerns. Its principle is to dissociate the implementation of various concerns constituting the software system to make its code more readable and more comprehensible. In this context, Kiczales et al. [114] explicitly define Aspect-Oriented Programming [114] by formalizing the concept of separation of concerns and define different steps to implement modular applications.

Aspect-Oriented Programming (AOP) is a programming paradigm not introduced to replace the other paradigms such as object oriented programming. It is rather an extension to the previous paradigms for supporting the modularization of concerns that cut across the implementation of a software application.

AOP introduces a unit of modularity, called *aspect* that implements a concern that would otherwise cut across several modules. The defined aspects are then woven (i.e. integrated) within the functional code using an aspect-oriented programming language to produce the

final application. AspectJ [113] is the most popular, stable sophisticated AOP language. AspectJ extends Java with new keywords to support the aspect concepts. Several aspect-oriented languages have later appeared but the majority of them is influenced by AspectJ. Since AspectJ can be considered as a standard for AOP in terms of language design, and for the sake of simplicity, we explain the concepts of AOP with an AspectJ example throughout this section.

The aspect encapsulates crosscutting concerns in complex systems by means of three key concepts: *join points*, *pointcuts*, and *advice*.

Join points are well-defined points in the execution of a program. The definition of these points depends on the join point model of the used aspect-oriented language. The join point model defines where, when and how aspects observe, augment or alter the program execution [9]. The joint point model influences the ability of an aspect oriented language to modularize the crosscutting concerns. In AspectJ, the join points correspond to the call of a method, the construction of an abject, the assignment to a variable, the execution of an exception handler, etc.

The pointcut is a query which picks out a set of join points, where some crosscutting functionalities should be executed. In addition, the pointcut can collect the context of those points. For example, a pointcut can select the call or the execution of a method in the base program, and capture the context of this method, such as the type of its parameters, the return value, the target object on which the method was called, etc. Other information can also be captured when the pointcut selects a set of related setter and getter execution points, a set of constructor calls, an exception hander, etc.

Each aspect oriented language defines pointcut constructs to capture the joint points. In AspectJ, the pointcut can contain wildcards (i.e. ".." and "$*$") to capture related set of join points and can use different predefined patterns to specify, for example, the fully qualified class name of the captured method or constructor, the type of the field, etc. The pointcuts can also be combined using the operators "&&" and "||" and can define a negation of a joint point using the operator "!".

The advice is a piece of code that implements a crosscutting functionality, which can be associated with pointcuts. The advice is executed whenever a join point identified by the pointcut is reached. The advice is similar to the method in object-oriented programming in some ways, but it is never called explicitly. The advice is always attached to a set of pointcuts which define in which points the advice code will be executed. The advice can be executed before, after, or instead of the join points that are selected by that pointcut, this corresponds respectively to the advice types *before*, *after* and *around* in AspectJ. With an around advice, the aspect can integrate the further execution of the intercepted join point in the middle of some other code. For example, in the advice of AspectJ language, the keyword *proceed* is used as place holder for the join point.

Furthermore, the advice has access to the information captured by the current pointcut. The data structures can be used in the code of the advice. For example, the AspectJ language provides constructs for extracting the arguments of a method call, the object which executes the intercepted method, etc. In addition to the advice and the set of pointcuts, the aspect can also contain helper methods which are used within the advice code.

Listing 2.1 presents an example of aspect in the AspectJ language. The aspect logs calls to objects of the class `Account`, which contains methods for crediting, debiting and transferring money.

```
1  public aspect Logging
2  {
3     //where?
4     pointcut logAccount(Account a) : call(* Account.* (..)) && this(a);
5     //when?
6     after (Account a) : logAccount(a)
7     {
8        //what?
9        System.out.println("The method " + thisJoinPoint + " is called" );
10       System.out.println("The new amount of this account is " +a.amount);
11    }
12 }
```

Listing 2.1: Example of aspect in AspectJ

In this aspect, the pointcut logAccount (line 4, Listing 2.1) specifies *where* exactly the logging concern will be executed. This pointcut captures the call of any method in the class Account without specifying the return types and the input parameters. The use of the wildcard "∗" serves to abstract from the return type and the name of the method, and the wildcard ".." serves to abstract from the parameters. The pointcut designator this is declared to identify the object *Account* that executes the intercepted methods.

The advice (lines 6–11) answers the questions about *when* and *what* behavior of the concern will be executed. This advice is associated with the pointcut logAccount declared with the instance a of the type Account (line 6). The advice will run after executing any joint point that is matched by the defined pointcut. In addition, this advice prints out two logging messages. The first one (line 9) specifies the method captured by the pointcut, using the keyword thisJoinPoint. The second message (line 10) uses the current object a, that executes the intercepted method, to show the variable *amount* defined in the class Account. In case the logging concern is required in other classes or modules, the developer can just modify or extend the pointcut. If this concern was not implemented as an aspect, the developer would have to localize all places where logging is required and add the corresponding code there.

Aspect weaving is a mechanism for integrating aspects with a base program. A weaver inserts the code of the advice at the appropriate places according to the associated pointcut definitions. There are two categories of weaving: static and dynamic.

Static weaving is performed by a tool, which takes the code of the base program and the aspects as input and outputs appropriate base code with the aspects integrated. Static weaving requires the analysis of the pointcuts defined in the aspect before adapting the code of the base program. The programmer cannot activate or inactivate the aspects once the program is compiled. This type of weaving allows for a good execution performance. Static weaving can also take place at load time instead of compile time. The weaver injects the aspect into the woven classes as they are loaded. AspectJ and CaesarJ [14] are two aspect-oriented languages that support static weaving.

Dynamic weaving occurs during the execution of the application. Therefore, the aspects can be deployed and undeployed at runtime. Dynamic weaving allows for a good reactivity and flexibility by adapting the base program to the changes of the requirements and environment. However, dynamic weaving can lead to large runtime cost when code needs to be woven or un-woven often. JAC [160] and PROSE [151] are examples of languages that support dynamic weaving.

Most of the aspect-oriented languages, including AspectJ, provide pointcut languages that are characterized [85] by a lack of expressiveness: only syntax patterns are matched and the available wildcards are not rich enough to easily express design requirements, which can give rise to long and awkward pointcut definitions that list many program locations. Such pointcuts are not robust, as they need to be updated consistently when the code or the API methods are changed.

Several research approaches and languages [32, 85, 118, 156, 196] have been proposed to enhance the expressivity of pointcut languages. These languages aim to define a robust and expressive semantic pointcuts by defining rich models of the program semantics and use abstraction mechanisms for defining the pointcuts. For example, different aspect-oriented languages [85, 118, 156] use Prolog as mechanism for defining an expressive pointcut language. Yang and Zhao [196] propose an approach for specifying pointcuts using program views, defined as abstractions of the classes and methods of the base program.

2.3 Specifying Non-functional Properties

Several approaches have been proposed for specifying and verifying non-functional safety properties. Some of them are generic (i.e., can be applied to any type of property) and others are applied only to a specific property and/or to a specific application domain. We classify these approaches into three groups: (i) approaches that propose UML profiles for specifying a specific non-functional property, (ii) aspect-oriented modeling approaches, (iii) approaches that use formal methods for specifying and verifying such properties.

For each group, we present, as an example, three related approaches to ours. Each approach allows specifying architectural properties, access-control properties, or temporal

properties. We already used these properties to validate our approach. More details on these properties and their applications are presented in the next chapters.

2.3.1 Using UML Profiles

UML profiles [153] are a standard extension mechanism for adapting the UML to particular domains (e.g. embedded systems, real-time applications, software architecture, etc.) and platforms (e.g. CORBA, EJB, J2EE). Extensions are defined using stereotypes, tagged values, and constraints which are applied on UML elements such as class, activity, connection, state, etc. Constraints are preferably specified using the OCL (Object Constraint Language). The OMG [153] standardized some UML profiles such as UML profiles for CORBA, for Enterprise Application Integration (EAI), for Schedulability, Performance and Time (UML/SPT). Several researchers further proposed interesting UML profiles for modeling different types of non-functional properties [42, 101].

Modeling architectural properties Hadj Kacem et al. [101] propose a UML profile for modeling dynamic software architecture of component-based applications. This profile is associated with three meta models to describe the architectural styles, the dynamic reconfiguration operations (i.e., add, delete, duplicate component/connection) specified in terms of pre- and post-conditions, as well as the reconfiguration protocol (i.e., the execution order of reconfiguration operations). This profile supports also the specification of architectural constraints that should be preserved during architecture evolution. The proposed profile is associated with new graphical notations by extending UML2.0 component and activity diagrams with new meta classes and stereotypes. This profile is implemented and integrated as a Fujaba plug-in [144].

Modeling security properties Cirit and Buzluca [42] propose a UML profile for integrating access-control policies in the early phases of software development. The proposed profile allows graphical modeling of role-based access control policies (RBAC) and their constraints such as static and dynamic separation of duties, cardinality and time constraints. The profile extends UML class diagrams with stereotypes representing the hierarchical RBAC concepts (e.g., role, role inheritance permission, user, session stereotypes, etc.). In addition to the informal description of each proposed stereotype, the authors define the basic RBAC rules and specify also tagged values for defining additional attributes and making relations between profile elements. The authors propose also the use of the OCL language to specify constraints to verify that the model defined by the designer is well-formed and does not violate the RBAC rules.

Modeling temporal properties In 2002, the OMG adopted a UML profile for Schedulability, Performance and Time (UML/SPT) [154]. This profile is based on a set of stereotypes and tagged values for modeling quality of service, concurrency, and real-time concepts.

It contains the sub-profile *RTtimeModeling*, which is associated with a meta model for representing time and time-related mechanisms. In addition, this profile supports predictive quantitative analysis of UML models. Based on this profile, designers can annotate sequence diagrams to specify, for example, the periodicity of an event, and/or the timing constraints on the actions of a component, etc.

2.3.2 Using Aspect-Oriented Modeling

Many aspect-oriented approaches have been proposed to aid the early stages in software development. Aspect-oriented modeling approaches allow software designers to isolate and separately address solutions for crosscutting concerns [183]. Different types of languages have been extended by aspect concepts. Some of these approaches are generic and can be applied for any crosscutting concern, and other approaches are proposed for specifying a specific property.

Most of the aspect-oriented modeling approaches are applied to UML, for modeling different crosscutting concerns such as security [78,159,166], temporal properties [96], software architecture [120]. These approaches focus on the specification of join points and pointcuts based on UML. They introduce aspect concepts using different graphical techniques, to weave the aspect based on the behavior of application instead of the inner details of the application.

Modeling architectural properties Krechetov et al. [120] provide a proper abstraction of the system using UML as visual notation to modularize architectural crosscutting concerns. They define the basic aspect-oriented modeling concepts and the requirements that should be considered for defining an aspect-oriented architecture modeling approach based on UML. The authors analyze and compare four existing aspect-oriented architectural design approaches and extract from them the most expressive and generic features (i.e., aspect concepts, such as advice, crosscutting interface, aspect-aspect and aspect-component relations, etc.). These features are combined in a single approach that satisfies the defined requirements.

Other approaches have been proposed by software architecture community to extend architecture description languages (ADLs) with aspect concepts. For example, Loukil et al. [132] propose an approach for specifying non-functional properties of embedded systems using AO4AADL, which is an aspect extension to the AADL. The AO4AADL specifications are then automatically translated to AspectJ aspects. Another interesting aspect-oriented ADL is called AO-ADL, proposed by Fuentes [163] for describing aspect-oriented architectures.

Modeling access-control policies Ray et al. [166] propose an approach for designing access-control policies using aspect-oriented modeling techniques. Access-control concerns

are specified as aspects that will be composed with a primary model (i.e., main functionalities of the application) to produce a complete design model of a secure application. The aspects are modeled as patterns for specifying (1) the structure of the policy (i.e., the entities constrained by the access control policy and their relationships) using stereotyped UML class diagram, and (2) the behavior of the policy (i.e., constraints that the access control policy imposes on behaviors) using stereotyped UML interaction diagrams. Ray et al. propose to model the hierarchical RBAC and static separation-of-duty policies as aspects and then compose it with the primary model.

Modeling temporal properties Iqbal and Elrad [96] propose an approach for modeling temporal behavior (i.e., time elapse and delays) using real-time statecharts. To model temporal properties in a modular way, the authors propose an extension of the aspect-oriented statechart framework [134] using time-annotated statecharts. This extension allows to model temporal properties (or any other crosscutting concerns) as separate statecharts and also allows to weave crosscutting and core concerns into a functioning system. As an example, the authors apply the extended framework to model the synchronization of local clocks with a global clock.

2.3.3 Using Formal Methods

Formal languages and methods are often used to rigorously define non-functional properties to avoid ambiguous descriptions that would be prone to mis-interpretation. Since formal methods are based on a formal foundation, several verification approaches and tools have been developed for verifying the specified properties, e.g., verifying that there is no conflict between specified constraints.

Software architectural properties have been specified using different formal languages. For example, Z notation [2], temporal logic [4], graph grammars [141], CSP [92], the π-calculus [155], etc. This diversity is due to the types of properties to be specified (e.g., structural, behavioral, and coordination properties, etc.), the types of architectures and styles (service-oriented architecture, component based architecture), and the abstraction levels that the designer is interested in.

Access-control policies can also be specified using various types of formal languages. Several approaches based on logics have been proposed for specifying role-based access control (RBAC) such as temporal logic [146, 172] or Z notation [1, 197]. Other formal methods have been also used to specify other access-control policies such as Petri Nets [174], graphs [119], process algebra [37], and automata [71, 130, 173]. The choice of the formal languages depends on the type of policies (e.g., RBAC, OrBAC, TBAC, Chinese Wall, etc.), and on the type of constraints. For example, the specification of separation-of-duties and delegation policies require temporal aspects, whereas the specification of access constraints on resources requires only a simple mathematical logic.

To specify temporal properties, researchers use different formal languages that support the concept of time. Temporal logic, timed automata and timed Petri nets are generally used to specify relative timed properties. These formal languages are also extended manually or formally to support the specification of absolute temporal properties using for example the XTUS language [35].

UML profiles are a specialization of UML for modeling domain-specific applications. This technique has appeared to resolve the limitation of UML for supporting different application domains. Similarly, Aspect-Oriented Modeling is an extension of the existing modeling languages (applied generally on UML) for specifying crosscutting concerns at a higher abstraction level. These techniques take advantage of the expressive power of standard visual notations provided by UML. The use of UML profiles and Aspects allows the designer to easily specify a specific non-functional property by annotating UML diagrams. In addition, several existing tools support the extension of UML in terms of plug-ins such as Eclipse UML2tools, Papyrus, Fujaba, etc. Despite the benefits of UML, there are some limitations to point out: (i) the absence of a clear and rigorous semantics of UML and OCL, (ii) the absence of formal verification tools to avoid any types of errors and contradictions, (iii) the limited expressive power of OCL for specifying complex and temporal constraints.

The formal languages allow rigorously specifying non-functional properties since they are based on mathematic foundations. Using a model checker and/or a theorem prover, the designer can also verify the specified properties. In addition, for each application domain there is at least an appropriate formal language that can be used to specify the corresponding non-functional properties. The choice of the appropriate formal language depends on the application domain, the type of property, the abstraction level, and also on the property to be verified. The designer can also combine two formal languages to cover all aspects of an application domain.

2.4 Runtime Verification Approaches

Runtime verification approaches allow to automatically check the behavior of a program against a set of expected properties. Generally, these approaches are composed of two main phases. The first phase consists in specifying the properties that should be verified, and the second one consists in automatically generating monitoring code from the high-level specification.

In this section, we present the most related approaches for verifying non-functional safety properties at runtime. First, we present an overview of the approaches that generate aspect code from UML models. Second, we focus on aspect languages that are used to verify at runtime non-functional properties using expressive pointcut languages. Third, we present

the Java-MOP framework, which allows formal specification of non-functional properties using different high-level specification languages and synthesizing runtime monitors. Fourth, we present two approaches that generate aspect code from a high-level specification of scenarios. Finally, we describe the three well-known runtime-verification frameworks *Java-MaC*, *Java Path Explorer*, and *Temporal Rover*.

2.4.1 Generating Code from UML Models

Several research approaches [23, 53, 83, 189] and commercial tools [94, 95] have been proposed to generate code in different programming languages from UML models. Some of them generate skeleton of code and others extend the UML to support the behavior of the application to generate executable code.

Most of these approaches are based on MDA (i.e., Model Driven architecture), which allows the designer to specify a platform-independent model (PIM) using an appropriate domain-specific language, generally, using a UML profile. Given a platform definition model (PDM), the platform-independent model is automatically translated to one or more platform-specific models (PSM) (e.g. a specific programming language, operating system, etc.). This mapping is defined based on well-defined transformation rules. This approach is also applied to generate aspect code from a UML model. In the following, we present two interesting works that generate aspect code from an aspect-oriented design model.

In [53], Cooper et al. propose a model-driven approach to automatically generate AspectJ-aspect stubs from an aspect-oriented design model. The proposed approach is based on three principal steps: First, the authors define an aspect-oriented design based on UML, as an extension to the formal design-analysis framework [55]. They propose to extend UML class diagrams with AOP concepts using icons, stereotypes, and new graphical elements. The designer uses the developed tool which supports these extensions to model his non-functional properties. Second, using a syntactic analysis, the visual design model is translated into XML. The generated file contains the required information to generate an aspect skeleton. Finally, the XML specification is automatically translated into AspectJ code stubs. The designer should complete the generated aspect with the respective advice to be integrated in the functional application code.

Groher and Schulze [83] propose an approach for specifying crosscutting concerns at the design level and generating the respective AspectJ aspects on the implementation level. The authors propose to extend UML with aspect concept without modifying its metamodel specification. This extension is based on the core concepts presented in AspectJ and supports a terminology where aspect and base elements are completely kept apart. In addition, the authors propose a validation module which allows the designer to validate the syntactical and semantical correctness of the design model, e.g. the validation of the existence of referenced pointcuts. After validating the model, the authors implement an automatic

mapping to translate the design models to concrete implementations in terms of AspectJ code skeletons.

One limitation of most MDA-based works is the lack of formal verification step to verify the consistency of the specifications and other properties related to the application domain, before performing the code generation.

2.4.2 Expressive Aspect Languages

Aspect-oriented languages are often used as runtime verification mechanisms. They intercept sensitive actions during the execution, and verify if the properties attached to this action are satisfied or violated. Several research approaches and languages [32, 60, 85, 118, 156, 196] have been proposed to enhance the expressivity of traditional aspect languages, such as AspectJ. These languages provide a high level specification (e.g., temporal logic, prolog, etc.) for easily intercepting sensitive actions by defining expressive pointcut languages.

J-LO: Java Logical Observer

J-LO [33,184], the Java Logical Observer, is a powerful runtime verification tool for checking temporal assertions in Java programs. It allows to verify temporal properties by building expressive formulae with free variables over AspectJ pointcuts. The tool uses an automata-based solution to verify if the execution of a program satisfies a high-level specification of temporal properties.

The workflow of using J-LO is shown in Figure 2.1 [184]. *First*, the designer should formally specify the temporal interdependencies between the application events using linear temporal logic (LTL). He specifies these properties (i.e., temporal formulas) as Java annotations in the source code of the program to be verified. *Second*, the annotated source code is compiled using a Java compiler. *Third*, J-LO extracts the annotations from the Java byte code and parses the temporal properties for generating the abstract syntax tree (AST). After that J-LO translates the temporal properties into an alternating finite automaton and generates also AspectJ aspects to trace the execution of the program. The generated aspects only monitor events related to the LTL specification. Based on the execution trace, the aspect browses through the generated automaton during the program execution to verify if the defined property is satisfied (entering the state representing "true") or detect violations by entering an error-state. *Fourth*, the generated aspects will be woven with the original program.

The generation process of J-LO is transparent, the designer specifies the temporal properties as Java annotations and the corresponding aspects are automatically generated and deployed in the original application code.

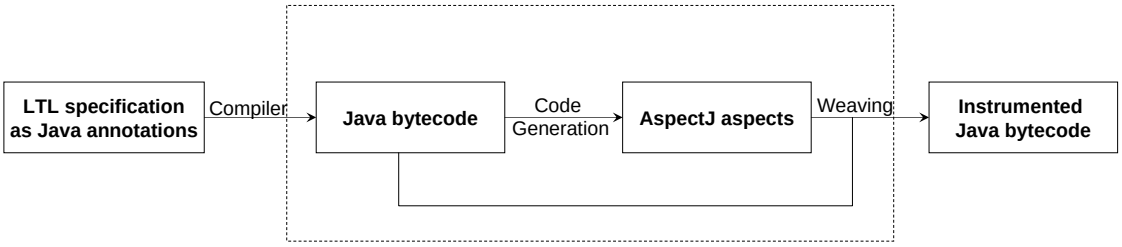

Figure 2.1: J-LO workflow

One limitation of J-LO is that no static verification mechanism is used to verify certain properties related to the specified formulas such as the absence of deadlock. In addition, J-LO does not verify the syntax and type errors of the properties before the compilation of the application source code. Another limitation of J-LO is that it is not efficient compared to other runtime-monitoring tools such as java-MOP [47] and tracematches [10].

Tracematches

Tracematches [10] is a history-based language for tracing and verifying at runtime the execution of object oriented programs by using, as J-LO, free variables in trace patterns.

Tracematches can be seen as an expressive and powerful pointcut language. The pointcut is specified via a regular expression, free variables, and a sequence of actions (defined as AspectJ joinpoints) to execute in a particular order during execution. The advice of this language, defined as an extra code, will be triggered whenever the regular expression matches a suffix of the execution trace (i.e., the defined order is satisfied) by taking into account the variable bindings.

The use of variable bindings in the pointcuts has two advantages: first, it allows the advices to access to context values at the joinpoints matched by the tracematch symbols. Second, it allows to match traces in the behavior of objects, rather than just control-flow traces, as defined in AspectJ.

Allan et al. propose a well-defined grammar and a declarative and operational semantics which is used to lead the implementation of Tracematches. The language is implemented as an extension of the AspectBench Compiler. The code-generation process is presented in Figure 2.2 (taken from [34]). First, the extension generates a finite state machine from the regular expression of each tracematch. Based on the generated automaton, this extension emits multiple Java classes which implement the actual runtime monitor that consumes these program events. Second, the extension generates an AspectJ aspect which triggers the finite automaton transitions to update its internal state when the relevant events occur at runtime. Finally, the compiler weaves the generated aspects into the program under test.

The weaving is performed in a transparent way, without generating the AspectJ file.

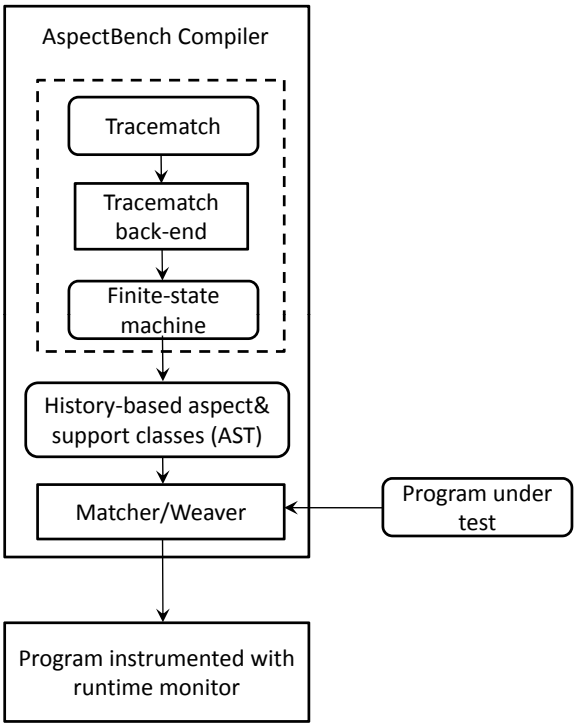

Figure 2.2: Tracematch workflow

Tracematches can be applied only for applications with a finite state. Another limitation of tracematches is that no verification step is proposed to formally verify that the regular expressions (defined as pointcuts) express a deadlock-free sequence.

2.4.3 Java-MOP: Monitoring Oriented Programming

MOP [46, 47] is a formal and extensible framework for runtime monitoring software and hardware systems. It can be applied to different programming languages such as Java and VHDL. Java-MOP [48,49], is an instance of the MOP framework for specifying and runtime monitoring safety properties in Java programs.

The current version of Java-MOP [140] supports several specification logics: the Java Modeling Language (JML), Extended Regular Expressions (ERE), Past-Time and Future-time Linear Temporal Logics (LTL), and Context-Free Grammar (CFG). These specification logics are integrated in a modular and extensible way, as logic plug-ins, which are seen as monitor synthesizers for properties defined as formulae. In addition, Java-MOP is not limited to these languages, the designer can also extends the Java-MOP framework by a

new domain specific specification formalism by providing logics plug-ins. He should also define the generation monitors of his formalisms inside the logics plug-ins. To ease the use of the proposed logics plug-ins, Java-MOP provides a general specification schema allowing the designer to specify his properties and define the system behavior when these properties are validated or violated.

Java-MOP separates the formal specification into two parts: the first part associates the interesting events and attributes to their corresponding Java methods declaration, while the second part contains the specification of the properties as a formula according to the underlying logic. To bridge the gap between the formal specification and the software implementation, Java-MOP automatically translates the formal specification into monitors and integrate it in appropriate places in the original program according to the configuration attributes defined by the designer. The code generation is encoded inside logics plug-ins. Java-MOP suggests the use of AOP paradigm and especially AspectJ as instrumentation mechanism. The AspectJ aspects are automatically generated and integrated with AspectJ compiler, as an extension of the original program.

Java-MOP is based on client/server architecture to easily extend the monitoring system with new formalisms such as logic plug-ins, to support various interfaces (web-based interface, a command-line interface, and Eclipse-based GUI) and for portability reasons.

For monitoring a complex property, Java-MOP generates several lines of code in the AspectJ aspect. The use of expressive pointcut language such as ALPHA [156] can facilitate the generation process from formal specification and allows also generating more comprehensible code.

2.4.4 S2A: for Scenarios to Aspects

Maoz and Harel [88] propose the compiler *S2A* to automatically generate aspects in AspectJ language from a Multi-Modal UML Sequence Diagrams (MSD). This formalism is a UML profile for specifying inter-object scenarios. Its semantics is based on that of Live Sequence Charts (LSC) which allows to specify various rich features such as the scenario that must (may or should never) happen.

They propose an approach for generating aspects to enforce the specified inter-object scenarios. The generation of aspects is based on a compilation scheme presented in [136]. Two types of aspects are generated: a set of scenario aspects and a single coordinator aspect. Each LSC specification is translated to a *scenario aspect* representing an automata. The scenario cuts are represented by automata states connected between them by the corresponding methods.

An AspectJ aspect is generated for each method specified in LSC scenario (i.e., automata transition). The pointcut intercepts this method and the after advice advances the

scenario aspect to the next state. The coordinator aspect collects the set of enabled and violated events and other information from the active scenario aspect. Based on the collected information and using a plat-out strategy, the coordination aspect chooses a method for execution, and ensures that this method will be executed by actual object.

In a recent work [137], Maoz and Harel propose an approach for analyzing, visualizing, and exploring the execution traces using a inter-object scenarios. These traces can be used for runtime monitoring, detecting errors that can be matched, and verifying coordination constraints during the execution of the scenarios.

2.4.5 M2Aspects

In [122], Krüger et al. propose *M2Aspects*, a tool which generates AspectJ aspect from a high level specification in Message Sequence Chart (MSC). They use an extended version of MSC to represent for example the alternative and the conditional repetition of messages and propose new operators such as the join operator to synchronize scenarios.

The main idea of the approach allows to explore different architecture candidates from a high level specification. The generated aspects are used to define prototypes for the different architecture candidates. These aspects implement the coordination of the interactions (i.e., defined as methods) between roles (i.e., an abstract concept representing components). The pointcut intercepts each specified method, and the after advice executes the next method as defined in the LSC specification.

This approach is similar to that proposed by Maoz et Harel [136]. In this work, the aspect executes the next method defined in the LSC scenario without taking in consideration the other scenarios that have the same method. In S2A, based on a defined strategy, the coordinator collects the required information and takes the decision on which method should be executed.

In [123], Krüger et al. propose a runtime verification approach. They generate aspect oriented runtime monitors from MSCs specifications.

The proposed approach uses a synthesis algorithm to translate the MSCs specifications into an optimized component state machine. This generated state machine is the input of the M2Aspect which generates the AspectJ aspects. These aspects identify and trace all exchanged messages between roles, and verify at runtime if the execution of the application is conform to the state machine (i.e., automatically generated from MSCs specifications). These aspects verify only the coordination of the interactions between roles. The pointcut intercepts the methods and the before advice shows an error message if the execution of the intercepted method is not conform to the specification.

The two previous works are similar, but they use just different languages and technologies. They propose a generation of AspectJ aspects from scenario-based software specifications. The generated aspects are used to implement the interactions between the entities constituting the system (i.e., object, components, etc.). They generate also history-based aspects to monitor at runtime the coordination of these interactions. In these approaches, the authors do not cover specifying the behavior of the entities constituting the system and their constraints.

2.4.6 Java-MaC

MaC [115] is a framework for runtime monitoring and checking software programs, independently of the programming language. Java-Mac [116] is a Mac prototype for Java.

The Mac architecture consists of two main phases: static phase and runtime phase. The static phase ensures the mapping between the high-level requirement specification and the low-level program implementation. In this phase, the designer manually specifies the safety properties (i.e. constraints) of the application inspired from the informal requirements specification.

The specification is composed of two parts. In the first one is the high level specification. Using MEDL language (Meta Event Definition Language), the designer specifies the invariants that should be satisfied during the system execution and associates them to their corresponding events. In this part, the designer is not interested in implementation details. The second part of the specification is called low level specification. Using the PEDL language (Primitive Event Definition Language), the designer defines the list of events and their corresponding method definitions that should be monitored during the system execution. According to the authors, the separation of the specification level and implementation level has two advantages: the first is to ensure that the Mac architecture can be used with any implementation language, and the second is to provide a clean interface to easily design and implement the design of the monitors.

In addition, the static phase generates three components, which are used, in the runtime phase to monitor and check if the specified invariants are satisfied.

The filter is the result of weaving the low level specification within the functional program. This filter keeps track of the changes of monitored objects and variables and sends these information to the event recognizer. Based on the received information, the event recognizer detects and recognizes the events according to a low-level specification and sends them to the runtime checker. The role played by the filter and the event recognizer can be combined in one component. Finally, the runtime checker verifies if the formally specified invariants are satisfied according to the current execution history.

In this approach, the role of the generated runtime components is limited to keeping track of the program execution and raise a signal if one of the specified invariants is not

satisfied. The generated components cannot enforce the specified invariants, and disallow the execution of the responsible method.

2.4.7 JPaX: Java Path Explorer

JPaX [89] is a runtime verification system for monitoring the execution of Java programs. JPaX is built on Pax, which is a generic environment to monitor programs in any programming language. JPaX is close similar to Java-MaC [115].

JPaX supports two kinds of verification. The first is the *logic based monitoring* that checks, during the program execution, the user requirements defined formally in a high level specification. These specifications are written in future and past temporal logic on top of Maude (i.e. a system based on rewriting logic and membership equational logic). The second kind of verification is the *error pattern analysis*, which consists in analyzing the trace of the event execution to detect error-prone programming practice such as deadlock and race detection.

The monitoring in JPax is performed in four steps: *First*, the designer formally specifies the requirement using temporal logic on top of Maude. Similar to the MaC framework, this specification is composed of two parts: one represents a high level specification and the other one is an instrumentation module which projects the specification to the program code. *Second*, The instrumentation module performs scripts to instrument the byte code of the application in order to be observed at runtime. *Third*, The instrumented program emits, at runtime, the event traces to the observation module. *Fourth*, the observation module, implemented in Java, verifies the error pattern analysis using defined algorithms.

To verify the logic based monitoring, there are two possibilities. (1) The observation module sends the list of execution events to the Maude program which evaluates them against the specified requirements. (2) The formal user requirements are translated to data structures and verified in Java.

Compared to the MaC framework, JPaX uses the Maude Program for formally specifying the user requirements. The designer can also specify other temporal logics on top of Maude and also verify them at runtime using JPaX. Besides the user requirements, JPaX is used to verify the various errors in the programs, such as the deadlock. The separation of the specification has several advantages (as cited in the Java-MaC section), but JPaX does not provide any mechanism to separate the monitoring module from the functional code at the implementation level.

2.4.8 Temporal Rover

Temporal Rover [61] is a commercial tool for runtime verifying programs written in different languages (e.g. C, C++, Java, etc.). It supports the linear temporal logic and metric

temporal logic, and two other specific operators for specifying temporal properties. These specifications are written as comments in the source code of the program. The designer should follow a precise assertion to specify his properties. He can also associate for each property, a list of actions which will be performed in special case, e.g., he can specify an action if the property is fulfilled or violated.

Temporal rover parses the source code and generates a new program that is similar to the original one. The specification of the temporal properties, defined as comments, will be automatically integrated in the source code of the new program. The algorithm for generating verification code is not presented.

Temporal rover does not provide any mechanism for checking the defined properties before the generation of code. In addition, the generated code is not well modularized.

2.5 Conclusion

After presenting background information pertaining to the proposed approach, this chapter detailed the related work on the specification and runtime verification of non-functional safety properties.

Through a classification of related work on non-functional safety properties, we concluded that formal methods seem to be the most interesting techniques for specifying and also verifying such properties. In addition, the study of different runtime verification approaches has shown two interesting results: The first one encourages the use of aspect-oriented programming as a runtime verification mechanism. The second one is that these approaches do not provide a generic and holistic process for specifying and enforcing non-functional properties.

In the next chapter, we will present the motivations of our work and we will detail our approach for specifying and enforcing non-functional safety properties. We will describe also the validation of our approach using three case studies.

Chapter 3
The Proposed Approach

3.1 Introduction

This chapter presents the motivations and an overview of our approach for implementing non-functional safety properties in distributed applications. In addition, this chapter presents the validation of this approach through three case studies.

The research works, that were presented in the previous chapter, proposed different solutions related to the specification, implementation, and/or the runtime verification of non-functional properties. These works are limited to detecting violations of the specified properties, they must enforce the properties instead. There is a lack of a complete methodology that supports all phases of the development process of these properties. In addition, most of the proposed approaches are interested in some specific properties in a specific application domain. They cannot be applied to other properties. Some of them use UML for modeling non-functional properties which lacks support for rigorous semantic analysis and lacks sufficient expressivity for specifying all aspects of these properties. Some other limitations are due to the gap between the specification and the implementation phases, the lack of formal foundations for designing non-functional properties, and the lack of appropriate mechanisms for modularizing such properties.

To solve these problems, we propose a holistic approach for implementing non-functional safety properties. Our approach allows formal specification, validation, formal verification as well as enforcement of these properties at runtime. Our approach is based on runtime enforcement, which is composed of two aspects:

(a) A formal specification of the properties to be verified at runtime. The formal methods are techniques useful for modeling complex software based on mathematical foundations. These techniques offer the possibility to build a rigorous specification with well defined syntax and semantics. Based on a rigorous specification, the designer can formally verify the properties of the system.

(b) Mechanisms to enforce the formally specified properties on particular program runs. Aspect-Oriented Programming is a programming paradigm that increases modularity by allowing the separation of cross-cutting concerns. We use this paradigm for the enforcement of non-functional properties. It allows to encapsulate the behaviors of these properties, which affect places in the application code, into reusable modules.

We validate our approach with three classes of properties. First, we apply this approach to structural properties of dynamic software architectures. Second, we focus on access control policies as an example of qualitative behavioral properties. Third, we apply our approach to temporal properties in Web-service compositions as an example of qualitative behavioral properties. In these application domains, we consider different types of architecture, formal methods and aspect languages.

The remainder of this chapter is organized as follows. Section 3.2 explains the motivations related to the implementation of non-functional properties. Section 3.3 presents an overview of the proposed approach and its three phases. In Section 3.4, we describe the validation of our approach through three applications (case studies). Section 3.5 concludes.

3.2 Motivations

Ensuring that software exhibits certain non-functional properties is crucial in many application domains. Static verification, while the most reliable means for this purpose, may not always be feasible. Runtime verification or enforcement is an alternative in such cases.

Specifying and enforcing these properties is important to ensure that bad things such as faults, deadlocks, etc. do not happen. Unfortunately, application designers and developers often focus on the functional concerns only and do not directly address non-functional properties including safety ones. Providing capabilities for the specification of various types of non-functional properties early in the development process, and their automated enforcement, are major challenges for designing reliable software systems. However, the state-of-the-art in addressing the specification and runtime enforcement of non-functional properties has several limitations. We will detail these limitations in the following sections.

3.2.1 Enforcing Non-Functional Properties

Most approaches presented in the previous chapter are focused only on the runtime verification of applications. They supervise the execution of the application at runtime and detect violations of the specified properties. These approaches generally specify the system as sequence of operations using a finite-state machine. The violation-detection is performed by verifying if the execution trace is conforming to the defined state machine.

Generally, just detecting the violation of a property is not sufficient and is unacceptable for critical applications. These approaches should not limit themselves to verifying the

specified properties at runtime, but they must enforce [68] the properties instead. Runtime enforcement avoids misbehavior by modifying the execution of the system, e.g. by skipping/stopping the execution of undesired operations.

3.2.2 Lack of a Holistic and Generic Approach

As discussed in the previous chapter, there are many approaches and tools that support several types of non-functional properties such as security, temporal properties, etc. But there is a lack of a complete methodology for implementing these properties. These approaches do not support all phases of the development process of non-functional properties. This process starts normally with (1) the specification of the properties using formal or semi-formal methods, (2) the verification of the specified properties, (3) the implementation or the generation of code that enforces these properties, (4) and finally, the integration of this code within the functional application code.
Non-functional properties are generally ignored in the early phases of the software development process of distributed applications. They are often considered to be identified and addressed at the implementation level, after the functional part is implemented, which can have a negative effect on the quality of the resulting application code.

Existing approaches are often focused on the specification and/or the implementation of non-functional properties for a specific domain. They cannot be generalized (i.e., they cannot be applied to other domains). In addition, many existing approaches focus on a single specific non-functional property, for example they propose a solution for implementing security policies, which cannot be applied to other properties.

3.2.3 Problems of Works using Non Formal Methods

Some existing approaches [23, 180] which are interested in the specification of the non-functional properties use informal or semi-formal languages such as UML as well as different extensions to it for specifying non-functional properties [102]. Such specifications have three problems. The specification cannot be verified formally and contradictions, in the specification cannot be excluded. Further, the OCL language, which is often used with UML for specifying constraints has a limited expressivity and cannot be used to specify complex properties such as separation of duties and delegation policies. Finally, the specification language may not cover all aspects of non-functional properties, for instance UML lacks means to express temporal properties.

Due to the lack of a formalism for identifying and formally verifying the non-functional properties, conflicts between these properties themselves and between these properties and the functional properties can happen in the modeling and the implementation phases.

3.2.4 Gap between Modeling and Implementation

There is a gap between the modeling and the implementation of non-functional properties. The code that checks and implements these properties is mostly written and integrated manually in the functional application code. Besides being a tedious activity, such a manual translation has two severe problems:

- No guarantee that the code implementing the non-functional property is conform to the high level specification. For example, the designer may forget to implement an important property or he may implement it more than once.
- Errors and contradictions may be introduced by the programmer when mapping a specification to code, although these errors do not exist in the high-level specification.

3.2.5 Crosscutting Concerns Modularity

In most existing approaches that are interested in the implementation of non-functional properties, the code corresponding to these properties is not encapsulated in separated modules. The implementation of these properties (concerns) cuts across the functional application code and cannot be cleanly modularized using conventional languages.

As a result, the implementation of non-functional properties becomes *scattered* across multiple and unrelated application modules and *tangled* with the functional code that implements the business logic. This lack of modularity leads to the following serious problems:

Problem in the change of the properties: Each change of a non-functional property requires the programmer to find all places in the implementation related to that property and update them accordingly. Such a process is tedious and error-prone when the code to be changed is not well-localized. This problem is accentuated especially because non-functional properties tend to change more frequently than the functional part of the application. For example, a change of the role based access control policy to another policy requires the update of several functional application modules, rather than updating a single module.

Difficult evolution: Since the non-functional properties are not implemented in separate modules, the programmer cannot easily addresses new non-functional properties because this requires reworking the implementation. In this case, the developer is limited to properties that have been already implemented at the beginning.

Low degree of code reusability: The developer cannot easily reuse a module that implements multiple concerns since the code corresponding to the non-functional properties is dispersed within the functional code. Other applications that require similar functionalities cannot reuse this module.

Poor code quality: If the code corresponding to the non-functional properties is not modular, the developer can produce hidden problems in the code, because he cannot concentrate an one module when he implements a non-functional property.

3.3 The proposed approach

We propose a holistic approach called SEVEN-pro for specifying and enforcing non-functional safety properties. It covers all phases of the development process of such properties: it starts by the specification using formal methods, then the validation and the formal verification of the specification, and finally the implementation of these properties and their integration in the functional application code.

SEVEN-pro combines the strengths of Aspect-Oriented Programming with those of formal methods. It allows the user to implement non-functional properties in an automatic, reliable and modular way.

On the formal methods side, we combine formal specification techniques to specify various kinds of non-functional properties. The choice of these techniques depends on the type of properties at hand and also on the considered application domain. For example, to specify temporal properties in Web service composition, we combine timed automata and XTUS language to specify respectively the properties related to relative and absolute time.

The high level specification of non-functional safety properties facilitates the tasks of the designer who generally concentrates on the specification of the main functionalities of his application. In addition, thanks to the formal verification and the associated tools, the designer can verify that his specification is not ambiguous and does not contain any contradictions.

In addition to the high level specification and verification of non-functional properties of distributed applications, our approach makes another important contribution: it presents a general scheme for automatically generating aspect code that enforces non-functional properties. Such generative approach, that compiles formal specification of non-functional properties into enforcement code, helps to avoid the risk of inconsistency at the implementation level.

Aspect-Oriented Programming has gained popularity as a programming paradigm that supports the modularization of crosscutting concerns. The crosscutting nature is a feature that aspects share with specifications of non-functional properties, which makes aspects a natural choice for compiling such specifications into code. We propose generic compilation schemes from formal specifications to aspect code. This process is automatic, whereby user input is expected for mapping a part of the specification to points in the implementation of the application.

Compared to other generative approaches that would compile formal specifications directly into code, the aspect-based generation has several advantages. The generation process is more direct and the resulting code is more reusable as it is well-modularized in aspects. Thanks to aspect generation, the approach as a whole can profit from advances in aspect-oriented language technology targeting faster aspect compilation time and runtime efficiency [30, 31]. Also, by being kept explicit, the crosscutting structure in which the non-functional properties manifest themselves in code becomes subject to advanced tooling technology for aspect-oriented systems. Examples are the AspectJ development tooling (AJDT), which makes the crosscutting structure of aspects explicitly visible in the development environment, and aspect-oriented refactoring technologies [124].

On the other side, compared to writing non-functional specifications directly as aspects [176], the combination of formal specifications and aspects has the advantage that the consistency of the specifications can be proved for correctness at the formal specification level, avoiding the risk of manually introduced errors and surprising aspect interactions at the code level [59].

The SEVEN-pro approach presumes a three-phase process, schematically shown in Figure 3.1.

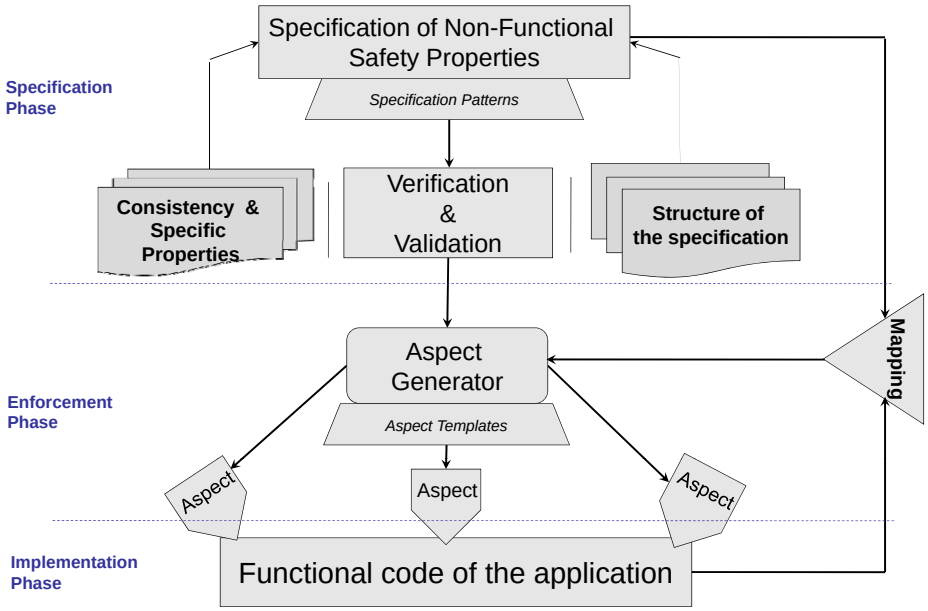

Figure 3.1: Overview of the SEVEN-pro approach

3.3.1 Formal Specification Phase

This phase consists in expressing at a high level the non-functional safety properties with the appropriate formal languages. It is composed of three steps.

In the first step, *the specification step*, the designer should formally specify the non-functional properties based on the defined specification patterns, whereby a specification pattern is a generalized description of a commonly occurring requirement [63]. The designer should instantiate and adapt the proposed specification patterns to the context of his application. He can also combine them to specify more complex properties.

For each considered application domain, we propose some specification patterns to cover the most relevant properties. We use these patterns for the following reasons:

- Patterns facilitate the specification of complex properties: The designer is not obliged to specify a complex property from scratch if this property is already defined as pattern.
- Patterns improve the reuse of specifications: The designer can instantiate, combine, or extend the patterns to specify the required properties. As an example, Blazy et al. [29] propose the reuse of specification patterns written in the B method (i.e., a formal method).
- Patterns structure the specification so that it can be easily translated to code. Without specification patterns, the designer can specify a property in different ways which may lead to problems in the automatic code generation.

For expressiveness reasons, we combine formal languages in order to cover the whole range of non-functional properties. For example, we use Z notation and Petri nets to specify, respectively, the architectural style and its reconfiguration operations, and the coordination protocol of dynamic software architectures.

In the second step, *the validation step*, the designer should validate his specification. To fully automate the code generation process from the high-level specification, the specification should be well structured, since it will be used as an input for a code generator.

The validation of the specification structure is performed according to the patterns presented above. This step allows to verify if the designer has correctly instantiated or combined these patterns. If the specification of the properties is not validated, the enforcement code cannot be automatically generated, and the designer should specify the properties again.

We use XML to validate the conformity of the specification with the proposed patterns. We represent patterns using the XML schema language (as a meta-model), whereas the specification of the non-functional properties is automatically translated to an XML document (as a model). Using the DOM package, the designer can easily verify the conformity

of the generated XML document to the XML schema and detect specification parts that do not conform to the defined patterns.

The third step, *the verification step*, plays a major role to ensure reliability. After the specification of the properties and the validation of the structure, a formal verification checks that the specification does not contain any contradictions between the specified properties.

Based on this rigorous model offered by the formal specification, performed as a first step, the designer can formally verify the properties of his system. He first verifies general properties such as the consistency, the absence of deadlocks, etc. and other properties related to his application. Second, he verifies application specific properties.

The designer can use two types of verification. The first one is *Model Checking*, used in the case that the system and its properties are modeled as a sequence (or set) of states connected by a set of transitions (i.e., labeled transition system). This model presents the chronology of the execution of the system (i.e., moving from one state to another). The designer can verify temporal properties expressed using a mathematical assertion language that can represent the system states over time like UPPAAL [27] or SMV [52]. The second type of verification is *Theorem Proving*, used when the system can be modeled as a set of mathematical definitions in some formal mathematical logic [13]. The designer should define a theorem for each property and then prove it using for example Z-EVES [139] (if the specification is written in Z notation as in our case), HOL [82], or ISABELLE [157].

For example, in the specification of the invariants of dynamic software architectures, we formally verify, using Z-EVES, the consistency of the specification, and the preservation of the system state after a reconfiguration.

3.3.2 Implementation Phase

In this phase, the developer provides the functional code of his application. This code contains only the business logic of the application and does not address any non-functional properties.

The functional code should conform to the formal specification of the non-functional properties in terms of the name, the type and the number of methods and their attributes. For example, if the formal specification states that a certain object has some attributes, then the class that implements that object should have fields that match these attributes. In the case that the functional code does not totally conform to the specification, the designer should create a mapping between the specification and the implementation. This mapping will be defined as an input for the code generator.

There is no restriction on the programming language and on the technology used for implementing the business process. For example, we apply our approach to a collaborative authoring system implemented as a Java-based application with the RMI technology. We

use also Web services with the BPEL language for implementing a Web service composition for a travel agency.

3.3.3 Enforcement Phase

The third phase ensures the relation between the two previous phases. It consists of the generation of aspect code from the formal specification defined in the first phase. The generated aspects will be automatically integrated, in a modular way, in the functional code defined in the second phase.

The generated aspects verify the specified properties at runtime. Before the execution of a critical operation in the main application (e.g., implemented as a Java Method), an aspect verifies all specified properties related to this operation, according to the system state. If these properties are satisfied, the operation will be executed, otherwise the aspect prohibits the execution of this operation and an exception is raised. For example, in a dynamic software architecture, before the execution of a reconfiguration operation that modifies the structure of the architecture, the generated aspects verify that the specified pre-conditions and the invariants related to this operation are satisfied.

The generation of aspect code from the formal specification is based on the transformation rules that we defined. These rules translate the specified constraints to code (for example in the enforcement of architectural properties, a Z constraint is translated to a Java conditional statement). These rules are based on well-defined aspect templates that define how to generate the corresponding pointcut and the associated advice. These templates facilitate and accelerate the code-generation process, and ensure more comprehensible code for developers. In some cases, the definition of the templates depends on the defined specification patterns, explained in the specification phase.

The pointcut of the generated aspect captures the execution of the critical operation, which is identified using the formal specification. The advice of the generated aspect has an *around* advice that is executed instead of the method calls to the critical operation captured by the pointcut. The around advice takes the decision to allow/prohibit the execution of the corresponding operation after monitoring the corresponding non-functional properties.

In our approach, there is no special restriction on the type of aspect that should be generated. An aspect compiler should be able to weave the generated aspects in the functional part of the application, as discussed in Phase 2. For example, in one case, we generate aspects in AO4BPEL, which is an aspect oriented extension for BPEL, for enforcing temporal properties in Web service composition. The generated aspects will be woven at runtime (without stopping the BPEL process) using the AO4BPEL engine. In another case, we generate aspects in ALPHA, an expressive aspect language extension to Java, for enforcing access-control policies.

3.4 Applications of Seven-pro

As shown in Figure 3.2, we apply SEVEN-pro to three different case studies to cover the different types of non-functional properties in distributed applications. In addition, we also diversify addressed the application domains, the types of architectures, as well as the formal methods and the aspect languages.

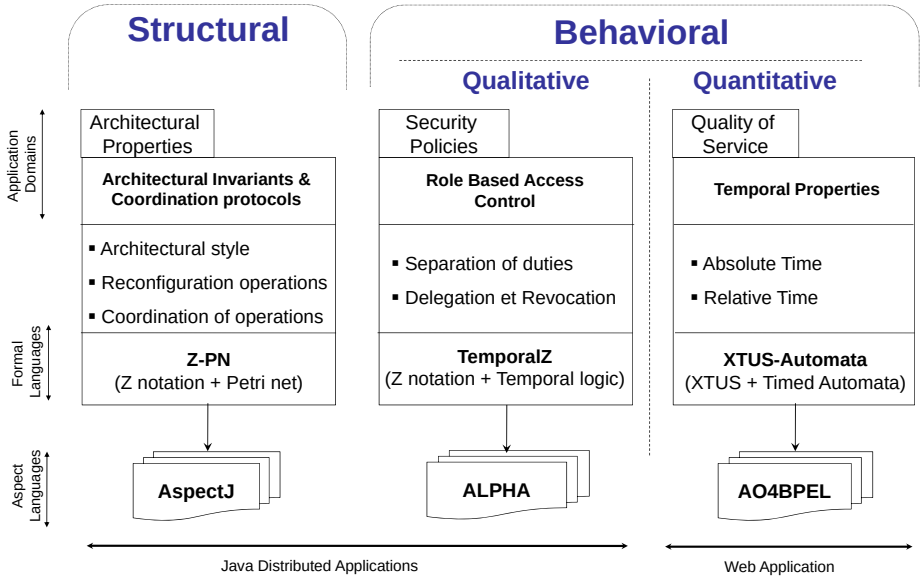

Figure 3.2: Application of the SEVEN-pro approach

3.4.1 The Types of Non-Functional Properties

We apply SEVEN-pro to both structural and behavioral non-functional properties. By *structural properties*, we mean those properties related to the structure and the architecture of the application, e.g., the cardinality of the objects constituting the system, the types of relation between objects. By *behavioral properties*, we mean those properties related to the behavior of the application, e.g., temporal constraints on certain operations of the application, or constraints on permitted operation sequences. We distinguish two kinds of behavioral properties: qualitative and quantitative properties.

In the context of *structural properties*, we specify the invariants and properties of dynamic software architectures. First, we specified the architectural style describing the types of objects constituting the system, the types of connections between these objects, and the

architectural constraints (i.e. invariants). Then, we specify the reconfigurations of software architectures, which correspond to the operations that modify the structure of the architecture. This specification includes also pre- and post-conditions. Finally, we define the coordination protocol describing the execution order of the reconfiguration operations. We formally verify the specification's consistency and the preservation of the system constraints after a reconfiguration.

With respect to *qualitative properties*, we apply the SEVEN-pro approach to security properties, specifically role-based access control. Using the *Temporal$\mathcal{Z}$* language, we formally specified the different types of static and dynamic separation of duties, which allow avoiding the risk of fraud in business processes by fine-grained control over the privileges for workflow tasks. We also specify several types and characteristics of delegation policies that allow the users to assign all or part of their permissions to other users.

For addressing *quantitative properties*, we apply the SEVEN-pro approach to temporal properties in Web services composition. We use XTUS-Automata, a formal specification language that allows to formally specify temporal properties related to absolute and relative time. We specify three types of temporal properties that can be combined to express more complex properties. The first type, *duration properties*, identifies the time between two BPEL activities. The second type, *temporal properties over cardinality*, identifies the number of occurrence of a BPEL activity in a defined time interval. The third type of properties, *absolute time properties*, expresses that an activity can be executed only within an absolute time interval.

Our approach supports also other properties than the previously mentioned ones. Software architectural properties, access control policy and temporal properties are juste examples. We can apply Seven-pro for example to other access control policies, resources related properties for embedded systems, etc.

3.4.2 The Types of Distributed Applications

We apply the SEVEN-pro approach to different types of distributed applications. In the first and second case studies, we focus on distributed Java applications that can be implemented using object or component technologies.

In the first case study, we validate our approach with a collaborative authoring system of structured documents based on client/server style. A set of writers and reviewers connect to a server to collaborate for editing shared documents. We enforce architectural properties of this applications to manage the collaborative edition of a document.

In the second case study, we consider a loan approval process as an example of a sensitive process with respect to security. This application has a critical nature, an access control policy is required to manage the duties and control the access of the bank employees to the process steps.

In the third case study, we are interested in service-oriented architecture. We look at Web service compositions implemented using the BPEL language. We consider a travel-agency scenario to book flights and hotels. The temporal properties have a high importance on this type of applications.

3.4.3 The Formal Methods and the Aspect Languages

In the three case studies, we try to use the most appropriate formal methods and aspect languages.

To cover all architectural properties in dynamic software architecture, we use both the Z notation and Petri nets to specify respectively the architectural invariants, pre- and post-conditions of reconfiguration operations, and coordination constraints. The specifications are automatically translated to AspectJ aspects. This aspect language ensures static compilation and weaving of aspects which is considered the most efficient manner to integrate aspects in the functional code.

Formal specification of the separation of duties and the delegation policies on top of the role-based access control requires the expressive power of the temporal logic as well as the first order predicates. For this reason, we use *Temporal$\mathcal{Z}$* as a formal language, which combines the linear temporal logic and Z notation. In contrary to the specification in the first case study, we do not use two separate formal languages, but we use *Temporal$\mathcal{Z}$* which embeds the temporal logic operators in the framework of Z notation: the temporal operators are used as Z operators. The *Temporal$\mathcal{Z}$* specifications are translated to ALPHA aspect code. This aspect language is characterized by an expressive pointcut language based on Prolog and it also supports the concept of time. The ALPHA aspects become effective only after they are deployed. The details and the advantages of this strategy are explained in [142].

For specifying temporal properties in a Web service composition, we require a formal language that supports both absolute and relative time. First, we defined the syntax and the semantics of the XTUS language using Z notation. Second, we propose a new language called *XTUS-Automata* that combines the XTUS language within Timed automata. Third, we propose an approach to specify the temporal properties of a Web service composition using BPEL. The formal specifications in XTUS-Automata are automatically translated to aspect code in AO4BPEL (aspect oriented extension for BPEL). These aspects can be dynamically woven within BPEL processes at runtime.

In our approach, we combine two formalisms for formally specifying different types of non-functional safety properties. Interferences between the formal specifications of each property that we supported cannot be happen, since we use complimentary formalisms for specifying different aspects of these properties. We do not propose to combine all supported properties in one application. We diversify addressed the non-functional properties to prove

the generality of our approach. In addition, we use different types of distributed applications. In the first and the second applications, we focus on distributed Java application, while for the third application; we look at Web services compositions using BPEL language.

3.5 Conclusion

In this chapter, we presented the whole approach for implementing non-functional safety properties in a reliable and modular way.

First, we described the motivations related to the development of these properties. We classified the motivations in four classes. The first one covers the motivations related to the lack of a holistic approach for non-functional properties, The second addresses the problems related to the use of non-formal methods. The third class presents the limitations caused by the gap between the specification and the implementation. The fourth class represents the motivations related to the crosscutting concern modularity.

Second, we presented the SEVEN-pro approach, which covers all phases of the development process for enforcing non-functional safety properties. This process is composed of three principal phases : (1) the formal specification of the non-functional properties, the validation and the formal verification of the specification, (2) the implementation of functional code, and finally (3) the generation of aspect code to verify at runtime the specified properties.

Finally, we presented the validation of the proposed approach through three applications. We explained also the choice of these applications and the diversity in terms of the type of properties, the type of architectures, the formal methods and the aspect languages.

In the three next chapters, we present the case studies used to validate the whole approach. In the next chapter, we focus on the specification and the enforcement of architectural invariants. In Chapter 5, we apply the SEVEN-pro approach to the separation of duties and delegation policies on the top of role-based access control. Chapter 6 describes the application of the approach to the specification and the enforcement of temporal properties in Web service compositions.

Chapter 4
Specifying and Enforcing Architectural Properties

4.1 Introduction

This chapter presents a concrete application of the SEVEN-pro approach to implement architectural properties. The contributions presented in this chapter were published in [105, 106, 111].

Some distributed applications are characterized by a dynamic architecture that evolves over the time to meet user requirements. For instance, new objects (or component) may be added, and existing connections between the objects may be modified during execution. When such reconfigurations are done, it is necessary to verify that no faults are caused and that the software application works correctly. Therefore, different architectural properties should be formally specified and also enforced to verify at runtime that each reconfiguration satisfies the respective properties.

We apply the SEVEN-pro approach for implementing architectural properties of object oriented applications in a reliable and modular way. As already mentioned in Chapter 2, this approach combines the strengths of formal methods with those of Aspect-Oriented Programming.

On the formal-methods side, we use Z [181] together with the Z-EVES tool [139] to specify and verify various kinds of architectural properties, and we use Petri nets [147] to model the coordination protocols. At the architectural specification level, formal methods are often used to specify architectural properties and to ensure their consistency by proving theorems on the specified architecture [90, 133, 141].

On the aspect side, we use the AspectJ [113] language to enforce the specified properties. Our approach automatically generates AspectJ aspects from the formal specification and integrates these aspects into the functional application code in a modular way. The aim of these aspects is to enforce properties of the architecture evolution at runtime. Each aspect intercepts the execution of a specific reconfiguration operation and allows or prohibits the

execution of this operation according to the satisfaction of the operation's pre-conditions, i.e., architectural properties.

The remainder of this chapter is organized as follows. In Section 4.2, we introduce the different architectural properties used in our approach. In Section 4.3, we describe an overview of the proposed approach and in Section 4.4, we present the case studies that we applied this approach to. Section 4.5 describes the formal specification of architectural properties and Section 4.6 details the aspect based enforcement of such properties. Section 4.7 presents an overview of our prototype. Section 4.9 reports on some related work and Section 4.10 concludes this chapter.

4.2 Software Architecture Properties

A software architecture is based on the principle of abstraction: hiding some of the details of a system through encapsulation to better identify and sustain its properties [175]. Consider the software architecture as an important issue for designing and developing complex software applications. A good specification of the architecture facilitates the development of applications that meet key requirements in terms of scalability, reliability and interoperability [74]. Taking into account these properties on the specification level (i.e., early steps of software development) allows their easy integration in the application and also may reduce the cost, the risk of error and duplication.

We are interested in applications that are characterized by a dynamic software architecture. Such an architecture evolves over time to meet user requirements. For instance, a developer could add a new element or delete an existing connection between elements during execution. We use architectural styles [75], one of the important concepts for specifying and structuring dynamic software architectures. Architectural styles describe families of architectures by laying down the component/connector vocabulary, the topology for structuring components and connectors, and constraints to be satisfied by any (re-)configuration of the architecture.

We focus on expressing and enforcing architectural constraints for distributed object-oriented software architectures. That is, the components in the subject systems are objects, whereby connectors can be method calls or events. More concretely, we target four kinds of semantic constraints:

1. Constraints on objects and object cardinality, e.g., constraints on the possible values of a certain object attribute, or on the total number of objects of a certain class in a system.

2. Constraints on object relationships and their cardinality, e.g., constraints on the number of objects of a class A that may be connected to an object of a class B. Using

this kind of constraints, the designer can specify, for example, the topology of the architecture.

3. Constraints on methods that modify the software architecture. These constraints represent pre-conditions and post-conditions that are defined before, respectively after, the respective method.

4. Method-call protocols, i.e., constraints on the order in which methods that modify the architecture should be called. For instance, "the method *m1* can be executed only after the methods *m2* and *m3* have been executed".

We call constraints of the first two types architectural invariants whereas we call constraints of the third type pre- and post-conditions of architecture reconfiguration operations because they constrain operations that modify the software architecture. We call constraints of the fourth type coordination constraints as they constrain the ordering of architecture-reconfiguration operations.

The execution of a method modifying the software architecture depends on the satisfaction of (1) the architectural invariants that are related to the input and output parameters of the method, (2) the corresponding pre-conditions, and also (3) the execution order of this method regarding the other application methods.

4.3 The Approach in a Nutshell

We applied the SEVEN-pro approach for specifying and enforcing software-architectural properties of distributed object-oriented applications as shown in Figure 4.1.

In the first phase, the designer formally specifies the architectural style of the desired application and the reconfiguration operations that describe the architectural evolution. The designer also defines the coordination protocol by specifying the execution order of the defined reconfiguration operations. After that, the designer can verify some generic and specific properties such as the consistency of the architectural style and the preservation of the style after a reconfiguration. The designer should modify the specification if one of these properties is not satisfied.

In the second phase, the developer provides the functional code of the object-oriented application. For now, we restrict ourself to Java code. This code provides only the core functionalities of the application and does not enforce any architectural properties. The functional code should conform to the formal specification of the components and their relations. For example, if the formal specification states that a certain object has some attribute, then the class that implements that object should have a field that matches that attribute. However, enforcing the formal specifications pertaining to individual objects is

out of scope for this work. Our focus is rather on the enforcement of architectural cross-object invariants.

In the third phase, the developer uses our aspect generator to generate a set of AspectJ aspects that enforce the formally specified properties. The advice is generated from the formal specifications, while the pointcut is generated from the functional application code. If the specification is not conform to the implementation, the developer should define a mapping between the formal specification of the reconfiguration operations and the implementation in terms of AspectJ pointcuts. This mapping will be used by the code generator.

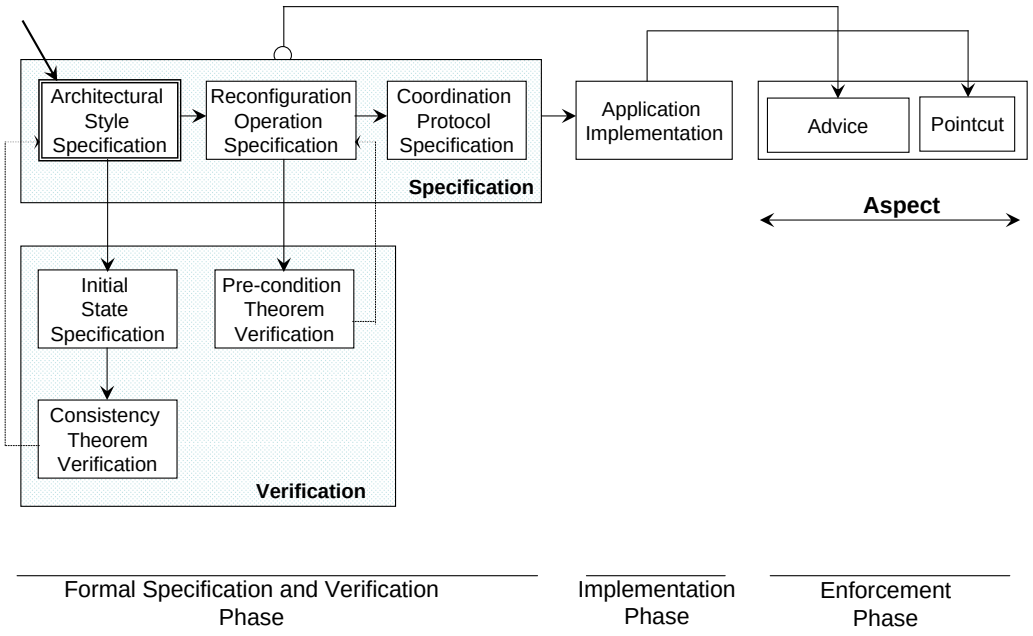

Figure 4.1: Specifying and enforcing architectural properties

4.4 Case Studies

As a proof of concept, we applied the proposed approach on two case studies with different architectural styles. The first is a collaborative authoring system for structured documents (CAS) [138] and the second is a patient monitoring system (inspired from [141]). For illustration, consider the class of software systems supporting collaborative authoring of structured documents. Typically such a system is organized in a client/server style. Documents are structured as a set of non-overlapping sections. These documents are located onto a server.

Clients connect to the server to view and edit these documents, at the granularity level of a section. Clients can have two different roles: *writers* can modify, create, and delete sections of a document, whereas *reviewers* can correct a section by adding annotations to it or by modifying the section formatting (i.e., they cannot edit the text but can change text fonts and colors). This application contains four distributed objects: shared document, section, writer and reviewer.

In this application, several important architectural invariants should be defined and enforced. For example, two writers should not be allowed to connect simultaneously to the same section. Another constraint is that the number of writers that are connected simultaneously to the same document should not exceed a predefined value. A third constraint is that a reviewer can only connect to documents that were modified by at least one writer since the last review. Other constraints affect the order in which re-configuration operations are executed. For example, a writer should not be able to edit a document that was changed by another writer before a reviewer corrects it.

Similar architectural invariants appear also in other applications with other architectural styles, for instance a patient-monitoring system defined according to the publish/subscribe architectural style. This system allows nurses to remotely control their patients within a certain hospital department. In this application, a nurse can see the patient's data, e.g., blood pressure, temperature, etc., by periodically receiving events from the respective bed monitor. Moreover, a bed monitor can raise an alarm to the responsible nurse if the patient's state becomes abnormal. In this application, several architectural constraints should be addressed. For instance, a nurse should not be assigned to more than five patients, otherwise, she will not be able to do her job appropriately. Further, to avoid having several nurses being called to the bed monitor when an alarm is raised, a patient should not be controlled by more than one nurse.

In the remainder of this chapter, we focus only on the *collaborative-authoring-system* case study and we present some examples in the specification and the enforcement of its architectural properties.

4.5 Formal Specification Phase

Formal methods are often used to specify dynamic software architectures and their properties. We used the Z notation together with the Z-EVES tool and Petri nets for specifying such properties .

Based on the classification of architectural properties presented in Section 4.2, we distinguish three kinds of specifications which should be defined and verified. (1) The specification of the structure of the individual objects as well as the architectural style, i.e., the overall architecture of the system, including the definition of the constraints on objects and objects

relationships. (2) The specification of the pre- and post-conditions for reconfiguration operations. The previous kinds of specifications are inspired from [133]. (3) The specification of valid sequences of reconfiguration operations. In the following, we present how formal languages, Z notation and Petri nets are suitable for describing these kinds of specifications.

4.5.1 Formal Languages

We use two well-known formalisms, namely Z notation and Petri nets, to specify architectural invariants and coordination protocols respectively. Using Z and Petri nets has two main advantages over using other formal specification methods as far as the particular needs of the kinds of architectural constraints that we address are concerned.

The combination of Z notation [181] and Petri nets [147], called Z-PN, allows us to cover all architectural properties previously presented in Section 4.2. Due to its set-theoretic and predicate-logic foundations, Z is well-suited not only for expressing constraints on the structure of individual objects and on relations between objects in an object-oriented system, but it can also be used effectively to specify the pre- and post-conditions of reconfiguration operations, which allows us to verify that reconfiguration operations do not break any architectural invariants. On the other side, Petri nets are well-suited to specify method protocols [3].

Z specification language

The Z notation [181] is a formal specification language for describing and specifying hardware and software systems (see Bowen's book [36] for some examples). Z was adopted as an ISO standard in 2002 [182]. The standard defines three languages: a mathematical language, a schema language and a refinement theory between abstract data types.

The mathematical language is based on set theory (i.e., set operators, set comprehension, cartesian products, and power sets) and on mathematical logic (i.e., first order predicate logic). Using the schema language, one can describe the state of a system and the manners according to which this state can change. Using the refinement theory, one can develop a system by building an abstract model from a system design.

A Z specification can be defined as a collection of state schemes and operation schemes. The state schema *State* describes the system state and the invariant relationships that are to be maintained when the system is updated. This schema consists of two parts: a declaration part and a predicate part. The latter defines constraints and specifies the values of the variables that are declared in the declaration part. The operation schemes *Operation* define the possible operations in the system, the relationship between their inputs and outputs, and the state changes resulting from their execution. An operation schema comprises the state *State* before and the state *State'* after performing the operation. These two states are represented in the schema language by $\Delta State$.

State
Declarations

Predicates

Operation
$\Delta State$
Input/output parameters

$pre - conditions$
$post - conditions$

Functions and constants are defined by an axiomatic definition as shown in the following form. The declarations introduce the type of the function or constant, and the predicates give its value. Every variable in Z has a particular type which can be defined as an abbreviation, a free type (i.e., user-defined type), an enumeration or a numeric type.

Declarations

Predicates

Researchers have proposed several tools for editing, animating and/or proving properties about Z specifications, such as ProofPower [15], HOL-Z [39], and Z/EVES [139]. We used Z/EVES for type and domain checking, schema expansion, precondition calculation, and theorem proving. For example, we verify the absence of contradictions between architectural properties and the preservation of the architectural style after reconfiguration. Z/EVES offers both graphical and command-line interfaces, and exports specifications to different formats including LaTeX, which we use as an input for the code generator.

Petri nets

Petri nets [147] are a graphical and mathematical tool to model and analyze discrete systems. In Petri nets, the different states of a system are modeled by *places* and *tokens*. Events are represented by *transitions* between places.
Formally, a Petri net can be defined as a 5-tuple $< P, T, F, W, M_0 >$, where:

- $P = \{p_1, ...p_m\}$ is a finite set of places,
- $T = \{t_1, ...t_n\}$ is a finite set of transitions with $(P \cap T) = \varnothing$ and $(P \cup T) \neq \varnothing$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
- $W : F \to \mathbb{N}_1$ is a weight function and
- $M_0 : P \to \mathbb{N}$ is an initial marking where for each place $p \in P$ there are $n \in \mathbb{N}$ tokens.

The system behavior can be described in terms of the system state and its changes. To simulate the dynamic behavior of the system, the state or marking will be changed according to the following rules:

- A transition t is enabled, if each input place p_i is marked with at least $W(p_i, t)$ tokens.

$$\forall\, p_i \in {}^\bullet t, M(p_i) \geq W(p_i, t) \tag{4.1}$$

where ${}^\bullet t$ denotes all input places for the transition t.

- If a transition t is enabled for the marking M then the enabling of t will lead to the new marking M':

$$\forall\, p_i \in {}^\bullet t, M'(p_i) = M(p_i) - W(p_i, t) + W(t, p_i) \tag{4.2}$$

where $W(P_i, t)$ is the weight of the arc (P_i, t), and $W(t, P_i)$ is the weight of the arc (t, P_i).

We use P3 [76] as a Petri-net software tool for modeling coordination protocols. P3 offers a graphical user interface to represent Petri-net models as graphs, and a simulator for the individual execution of transitions and parallel execution of all transitions. P3 exports Petri-net models to different formats including XML, which our code generator uses as input.

4.5.2 Specification and Verification of Overall Architecture

Predicate logic is used to specify properties of the individual objects participating in the system following the object specification pattern shown below. In this pattern, att_i denotes an attribute, pr_i denotes the properties of an object.

$$\begin{array}{|l} \textit{Object}_i \\ \hline att_1 : \textit{Type}_1, att_2 : \textit{Type}_2, ..., att_n : \textit{Type}_n \\ \hline pr_1, ..., pr_n \\ \hline \end{array}$$

As an example, consider two kinds of objects in our collaborative authoring system: shared documents and sections. A shared document is accessible to any client that is authorized either as a Writer or as a Reviewer. A section is defined by the positions of its first and last characters in the whole document. In the schema below, the shared document is defined as a sequence of sections that do not overlap, as specified by the predicate in the lower part of the specification of a shared document.

$$\begin{array}{|l} \textit{Section} \\ \hline \textit{firstCharacter} : \mathbb{N} \\ \textit{lastCharacter} : \mathbb{N} \\ \hline \textit{lastCharacter} \geq \textit{firstCharacter} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{SharedDoc} \\ \hline \textit{sections} : \operatorname{seq} \textit{Section} \\ \hline \forall\, i : \mathbb{N} \mid 1 \leq i < \#\textit{sections} \\ \quad \bullet\, (\textit{sections}(i+1)).\textit{firstCharacter} \\ \qquad = (\textit{sections}(i)).\textit{lastCharacter} + 1 \\ \hline \end{array}$$

The overall system specification defines a set of objects, the relationships between them, and the architectural invariants that must be maintained when the system evolves. In the sample *System* schema shown below, o_i denotes an object instance, $Object_i$ denotes an object type, $relation_{ij}$ denotes a relation between $Object_i$ and $Object_j$ (as denoted by the bidirectional arrow), and Apr_i denotes architectural invariant. These invariants describe constraints on objects (e.g., the maximum and/or the minimum number of the object instances, etc.), and constraints on object relationships (e.g., the domain and the range of the defined relations, the maximum and/or the minimum number of the relation instances, etc.).

System

$o_i : Object_i$;...
$o_j : \mathbb{F}\ object_j$;...
$relation_{ij} : object_i \leftrightarrow object_j$;...

$Apr_1, ...Apr_n$

To verify that our architectural style does not contain any contradiction within the defined invariants, we verify the system consistency by ensuring that at least one valid initial state exists [192]. Using Z/EVES, we can define an *InitialisationTheorem* and prove it, whereby

- *System* represents the system schema that describes the architectural style
- *SystemInit* corresponds to a Z schema that describes the initial system state.

SystemInit

System

$o_i = \{...\}$
$R_ij = \{(..., ...), (..., ...)\}$

Theorem *InitialisationTheorem*
$\exists\ System \bullet SystemInit$

For illustration, the system *CollaborativeAuthoringSystem* is specified in the schema below. It consists of finite sets ($\mathbb{F}$) of writers and reviewers, a shared document and relations between authorized writers/reviewers and sections of the shared document. Conditions on the relations are preserved by verifying the domain *dom* and the range *ran* of each relation (as defined in [C1]). Some other invariants are given: [C2] limits the number of the writers to six. [C3] specifies the maximum number of writers that are connected to sections and [C4] fixes the number of occupied sections by reviewers or writers to four. [C5] states that a writer or a reviewer can be connected to only one section at any point in time. [C6] states that two actors (writers or reviewers) are never connected simultaneously to the

same section. These constraints should be satisfied by any operation that changes the sets of writers, reviewers, and sections.

CollaborativeAuthoringSystem

$$
\begin{array}{l}
writers : \mathbb{F}\ Writer \\
reviewers : \mathbb{F}\ Reviewer \\
sections : \mathbb{F}\ Section \\
WriterSection : Writer \leftrightarrow Section \\
ReviewerSection : Reviewer \leftrightarrow Section \\
\hline
\mathrm{dom}\ ReviewerSection \subseteq reviewers \qquad [\mathsf{C1}] \\
\mathrm{ran}\ ReviewerSection \subseteq sections \\
\mathrm{dom}\ WriterSection \subseteq writers \\
\mathrm{ran}\ WriterSection \subseteq sections \\
\#writers < 6 \qquad [\mathsf{C2}] \\
\#reviewers < 5 \\
\#sections < 7 \\
\#(writers \lhd WriterSection) < 4 \qquad [\mathsf{C3}] \\
\#(reviewers \lhd ReviewerSection) < 3 \\
\#((ReviewerSection(\!|\ reviewers\ |\!)) \qquad [\mathsf{C4}] \\
\quad \cup (WriterSection(\!|\ writers\ |\!))) < 4 \\
\forall\, w : writers \bullet \#(WriterSection(\!|\ \{w\}\ |\!)) \leq 1 \qquad [\mathsf{C5}] \\
\forall\, r : reviewers \bullet \#(ReviewerSection(\!|\ \{r\}\ |\!)) \leq 1 \\
\forall\, r : reviewers; w : writers; s : sections \\
\quad \bullet\ (w, s) \notin WriterSection \lor (r, s) \notin ReviewerSection \qquad [\mathsf{C6}] \\
\forall\, r, rr : reviewers; s : sections \mid r \neq rr \\
\quad \bullet\ (r, s) \notin ReviewerSection \lor (rr, s) \notin ReviewerSection \\
\forall\, w, ww : writers; s : sections \mid w \neq ww \\
\quad \bullet\ (w, s) \notin WriterSection \lor (ww, s) \notin WriterSection
\end{array}
$$

To verify the consistency of the *CollaborativeAuthoringSystem* schema, we define an initial system state, *InitCASystem*, shown below. The initial state consists of two writers $w1$, and $w2$, one reviewer $r1$, and three sections $s1, s2, s3$. The proof of the consistency theorem ensures that the specification of our collaborative authoring system is consistent and does not contain any contradictions.

InitCASystem

$$
\begin{array}{l}
CollaborativeAuthoringSystem \\
\hline
writers = \{w1, w2\} \\
reviewers = \{r1\} \\
sections = \{s1, s2, s3\} \\
WriterSection = \{(w1, s1), (w2, s3)\} \\
ReviewerSection = \{(r1, s2)\}
\end{array}
$$

Theorem *ConsistencyCASystem*

$$\exists\, CollaborativeAuthoringSystem \bullet InitCASystem$$

4.5.3 Specification and Verification of Reconfiguration Operations

Each reconfiguration operation is specified by means of a Z *operation schema*, which defines the input parameters ($c_i?$) as well as the pre- and post-conditions. These conditions are essential to verify that the evolution of the architecture preserves the defined invariants. Reconfiguration operations are executed only if their pre-conditions are satisfied. In the operation schema below, $PreCond_i$ and $PostCond_i$ denote a pre- and a post-condition of the reconfiguration operation $Operation_i$.

$$
\begin{array}{|l}
\hline
Operation_i \\
\Delta System \\
o_i? : object_i; \\
... \\
\hline
PreCond_i, ..., PreCond_n \\
PostCond_i, ..., PostCond_n \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{Theorem}\ PreConditionTheorem \\
\quad \forall\, System, o? : Object_i \\
\quad\quad \mid preConditions(System, o?) \\
\quad\quad\quad \bullet\ \text{pre}\ Operation_i
\end{array}
$$

After specifying the reconfiguration operations formally, these operations need to be verified. To evaluate the impact of a reconfiguration operation on invariants, we define and prove the pre-condition theorem *PreConditionTheorem* shown on the right-hand side of the specification above, whereby (as defined by [128]):

- *System* denotes the Z schema describing the architecture style (see previous section).
- *o?* denotes the input parameter presenting the objects related to this operation.
- *preConditions(System,o?)* denotes the pre-conditions in terms of the defined *system* and the input parameters *o?*.
- $Operation_i$ denotes the Z operation schema describing this operation.
- **pre** $Operation_i$ describes that a final system state denoted by $system'$ (i.e., the new system state after the execution of the current operation) with the output parameter (out !) can be shown to exist. Formally,

$$\exists\, System'; out! : Output \bullet Operation_i$$

This theorem specifies the pre-conditions that must initially be satisfied to guarantee that the constraints are preserved after the execution of the operation and verify that the execution of the reconfiguration operation preserves the architectural style describing the whole system.

Let us now illustrate the approach to specify reconfiguration operations by means of our collaborative authoring system. We have specified and formally validated reconfiguration operations such as the insertion and connection of writers, reviewers, and sections. For

illustration, the following schema specifies the operation *ConnectWriter*. As pre-conditions, the operation schema states that when a writer $w?$ is connected to a section of the shared document, then it should be one of the writers that are already present in the system ([preC1]) and the section $s?$ should be already created ([preC2]). As post-conditions, the relation *WriterSection* is modified. After the execution of the operation *ConnectWriter*, the new state of the relation denoted by $WriterSection'$ contains additionally the couple composed of the new writer $w?$ and the new section $s?$ ([postC5]). The other objects and relations defined in the architectural style are not modified after the execution of this operation ([postC1, postC2, postC3, postC4]).

ConnectWriter

$w? : Writer$
$s? : Section$
$\Delta CollaborativeAuthoringSystem$

$w? \in writers$	[preC1]
$s? \in sections$	[preC2]
$reviewers' = reviewers$	[postC1]
$writers' = writers$	[postC2]
$sections' = sections$	[postC3]
$ReviewerSection' = ReviewerSection$	[postC4]
$WriterSection' = WriterSection \cup \{(w?, s?)\}$	[postC5]

To validate the connection operation of a new writer, we use Z/EVES to prove the theorem *PreConnectWriter*, which ensures that the connection of a writer conforms to the system invariants described in the system schema *CollaborativeAuthoringSystem*. The theorem below states that the connection of a writer to a section requires that no reviewer or any other writer are connected to that section.

Theorem *PreConnectWriter*
$\forall CollaborativeAuthoringSystem;$
$w? : Writer; s? : Section \mid w? \in writers \land s? \in sections$
$\land (\forall rr : reviewers \bullet (rr, s?) \notin ReviewerSection)$
$\land (\forall ww : writers \bullet (ww \neq w? \land (ww, s?) \notin WriterSection))$
$\bullet$ pre *ConnectWriter*

4.5.4 Specification of Valid Protocols

We use Petri nets to define constraints on the execution order of reconfiguration operations that are already specified in Z. Typical coordinations in distributed object-oriented

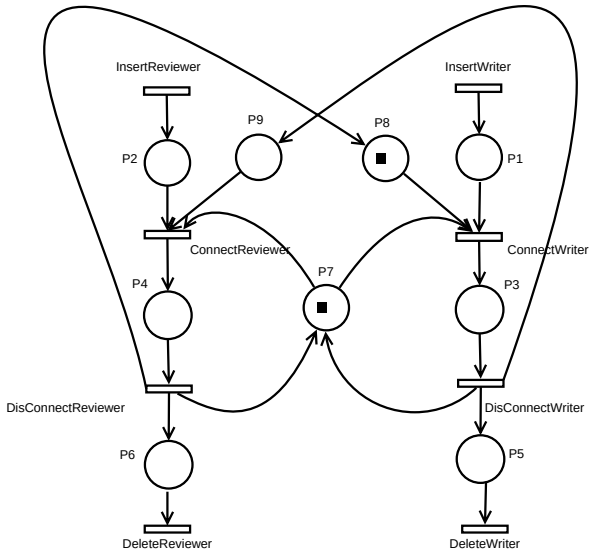

Figure 4.2: Valid protocol of the collaborative authoring system

applications such as synchronization, mutual exclusion, parallelism, etc., can be naturally specified with Petri nets. We model each reconfiguration operation by a transition and the system state by a set of places and tokens. Enabling a transition in the Petri net means that the corresponding reconfiguration operation conforms to the constraints given the current system state. Consequently, the transition can be executed and its target places can be occupied by tokens.

In our collaborative authoring system, the writers can create, modify, and delete sections. Then, the reviewers can correct these sections and add annotations. To enforce the activity order described above, we define a coordination protocol that requires that each section must be created or modified by a writer before it becomes accessible to reviewers for correction. In addition, after a section is corrected, the next reviewer cannot revise it before an author modifies it.

In the initial state shown in Figure 4.2, the transitions *InsertWriter* and *InsertReviewer* are always enabled. Consequently, the transition *ConnectWriter* will be enabled. Thus, a writer can connect to a section. After enabling the transition *DisconnectionWriter*, the writer can be deleted but it cannot connect because there is no token in $P8$. However, a reviewer can still connect because the transition *ConnectReviewer* is enabled.

4.5.5 Verification of Specification Structure

To facilitate the automatic translation of Z specifications to aspects, we proposed specification patterns that describe how to structure formal specifications of software architectural properties. We do not define any patterns for Z predicates. The designer can define any Z predicate to express the architectural constraints. He should only respect the following rules in his specification.

1. Each object must be specified by a Z schema that defines a unique identifier for the object and one or more constraints on the object.
2. An object can be atomic or composite (composed by a set or a sequence of sub-objects).
3. The architectural style (i.e., presenting the whole system architecture) must be specified by a Z schema.
4. The architectural style consists of a finite set and/or a sequence of object instances. Each object must be connected to at least one other component.
5. Object connections must be specified using Z relations ($\leftrightarrow$) and the domain (dom) and range (ran) of each relation must be specified in the system schema.
6. The reconfiguration operations must be specified using Z operation schemes. This schema consists of a set of input parameters presenting the related objects, the preconditions and the post-conditions in terms of defined objects and relations.
7. Each Petri-net transition should correspond to a reconfiguration operation already specified by a Z operation schema.

These rules are specified as a meta model, using XML Schema Definitions. To verify that the Z-PN specification is compliant with this meta-model, we translate the specification into an XML document. We use the Java package JDOM to verify that the XML representation of the Z specification satisfies the meta model.

4.6 Mapping Formal Specifications to Code

This section describes how formal specifications are mapped to code. The mapping of the specifications pertaining to individual components and to the overall system architecture are done manually. We focus on mapping cross-component specifications related to reconfiguration operations and their valid protocols. This mapping is done by translating specifications to aspects in the AspectJ language. In the following, we first present the structure of the generated aspects and subsequently describe the code generation process.

4.6.1 Aspects that Implement Cross-Component Invariants

The layer that is responsible for checking and enforcing architectural constraints at runtime consists of a set of aspects that are generated from Z operation schemes and corresponding Petri-net transitions. There is one aspect per reconfiguration operation, i.e., per each pair of Z operation schema and corresponding Petri-net transition.

For example, consider again our collaborative authoring system. Each reconfiguration operation, e.g., insert writer, connect reviewer, delete section, etc., corresponds to a Z operation schema in the formal specification of the architectural style and to a Petri net transition in a coordination protocol. For each reconfiguration operation, an AspectJ aspect is automatically generated. In other words, the aspects connect the formal level and the functional level to ensure that formally specified invariants are maintained when the architecture evolves.

The overall structure of these aspects is as follows. The pointcuts of the generated aspects intercept the execution of operations from the functional layer that correspond to architecture reconfiguration operations. For example, every call to the operation `insertW` in the public interface of the class `Writer` corresponds to the reconfiguration operation *InsertWriter*. The advice of the generated aspect first checks and enforces the coordination protocols that are defined by Petri net transitions and then checks the constraints specified in Z. The first check verifies whether the transition in the Petri net is enabled, based on the current system state. The second check verifies the constraints and pre-conditions corresponding to the reconfiguration operations that are specified within the style schema and the operation schema. If both checks are successful, the aspect executes the reconfiguration operation, and after that, the aspect updates the state of the system. Otherwise, the aspect prohibits the execution of the reconfiguration operation.

For illustration, consider the aspect shown in Listing 4.1, which controls the connection of a writer to a section. This operation is specified by the Z operation schema *ConnectWriter* and the Petri-net transition with the same name. The pointcut `ConnectWriterToSection` (lines 3 and 4 in Listing 4.1) selects all calls to any public method *connectW* independently of its parameters. The constructs *target* and *args* are used to bind to the variables `w: Writer` and `s: Section` respectively the target, respectively the arguments, of the method-call joinpoints that the pointcut selects.

The around advice of the aspect in Listing 4.1 controls the execution of the reconfiguration operation `ConnectWriter`. First, the advice executes code to ensure that the coordination protocols are respected. For example, in the Petri net shown in Figure 4.2, we specified that a writer cannot modify a section unless a reviewer has corrected it. The generated advice contains a call to the method `isTransitionEnabled` (line 9) to check whether the transition `ConnectWriter` is enabled, given the current state of the system. Next, Z constraints are checked.

```
1  public aspect EnforceConstraintsForConnectWriter {
2
3      pointcut ConnectWriterToSection(Writer w,Section s):
4                      call(public * connectW(..)) && target(w) && args(s);
5
6      void around(Writer w, Section s): ConnectWriterToSection(w,s)
7      {
8          ...
9          if (isTransitionEnabled (CurrentMarking, ConnectWriterTransistion)) { ...}
10
11         ...
12         if (isMemberOf(wInput,SystemState.writers)) { ... } //Pre-condition preC1
13         if (isMemberOf(sInput,SystemState.sections)) { ... } //Pre-condition preC2
14         ...
15
16         if (checkConstraintC1()) { ... }
17         if (checkConstraintC2()) { ... }
18         ...
19
20         if (allConstraints) {
21            proceed(w,s);
22            updateSystemState(w,s);
23          }
24          else { ... }
25        }
26
27     boolean checkConstraintC1() { ... }                //Constraint C5
28
29     boolean checkConstraintC2() {                      //Constraint C6
30          boolean constraints, result0, result1, result2 = true;
31          String [] Tab0 = SystemState.Newreviewers;
32          while ((Tab0.length!=0) && result0) {
33              String r = getFirstElement(Tab0);
34              String [] Tab1 = SystemState.Newwriters;
35          }
36          while ((Tab1.length!=0) && result1) {
37              String w = getFirstElement(Tab1);
38              String [] Tab2 = SystemState.Newsections;
39          }
40          while ((Tab2.length!=0)&& result2) {
41              String s = getFirstElement(Tab2);
42              constraints = Or(isNotMemberOf(w, s,SystemState.NewWriterSection),
43                               isNotMemberOf(r, s,SystemState.NewReviewerSection));
44              result2 = result2 && constraints;
45              result1 = result1 && constraints;
46              result0 = result0 && constraints;
47           }
48           return result0;
49      }
50
51      void updateSystemState(String NameW, String NameS) { ... }
52 }
```

Listing 4.1: Example of the generated AspectJ aspect

For example, the constraints [preC1] and [preC2] from the Z operation schema `connectWriter` are translated to calls to the method `isMemberOf` of a Z-operator package that we implemented (lines 12 and 13 in the Listing 4.1). Quantified constraints such as the constraint [C5], which ensures that a writer or reviewer can modify only one section at a given point in time, and the constraint [C6], which disallows overlaps between sections, are translated to Java code by using helper methods. For instance, a helper method `checkConstraintC2` (lines 29–49) is generated to evaluate the constraint [C2]. This helper method is called in the advice (line 17).

Moreover, the advice contains a method `updateSystemState` (lines 51) that updates the system state that is stored in a generated class, which represents the components of the system, their relationships, and the marking of the Petri net. If all generated constraints are evaluated to **true**, the reconfiguration operation will be executed (using *proceed*) and the system state will be updated (lines 20–24).

4.6.2 The Aspect Generation Workflow

The aspect generation workflow is composed of two main phases. The first phase consists in generating the AspectJ pointcut while the second consists in automatically translating Z and Petri net specifications into an around advice associated to the generated pointcut.

The generation of pointcut can be performed manually or automatically. If the specification of architectural properties is conform to the implementation of the functional code, the pointcut will be automatically generated, i.e., without the intervention of the developer. It will be generated from the Z operation schema representing the corresponding reconfiguration operation implemented as a Java method. In the case that the specifications of architectural properties are not totally conforming to the implementation, the developer should provide an AspectJ pointcut for each specified reconfiguration operation, denoted in a XML file. These pointcuts ensure the mapping of the reconfiguration operations to points during the execution of the functional code. For example, Listing 4.2 shows the mapping of the reconfiguration operation `ConnectWriter`, defined in the formal specification, to calls to the method `connectW` (lines 5–7 in the Listing 4.2) from the functional level.

```
1  <Mapping Name="CollaborativeAuthoringSystem">
2  <PointCut Reference="InsertWriter">
3        InsertW(Writer w):
4           call(public * insertW(..))&& target(w); </PointCut>
5  <PointCut Reference="ConnectWriter" Component="Section">
6        ConnectWriterToSection(Writer w,Section s):
7           call(public * connectW(..)) && target(w) && args(s); </PointCut>
8  ...
9  </Mapping>
```

Listing 4.2: Mapping formal specification to code

The generation of advice consists in translating Z predicates and the enabling of Petri transitions to Java condition code. The aspect code generator extracts the properties of the objects, the relations between objects, and the architectural constraints that are specified in the system schema and in the operation schemes. For each reconfiguration operation, which corresponds to a Z operation schema and to a transition in the Petri net, the pre-conditions are translated to pieces of around advice as schematically shown in line 5 of Listing 4.3. The generated advice is associated with a pointcut that can be automatically generated or provided by the programmer in the XML mapping file (cf. line 2 in the Listing 4.3).

The workflow of the generated around advice is composed of three phases. First, the advice checks whether the Petri net transition for the corresponding reconfiguration operation is enabled. For that purpose, we automatically generate a call to the method `isTransitionEnabled` that take in parameter the current marking and the name of the Petri nets transition (line 8 in the Listing 4.3). This information is extract from an XML representation provided by the P3 tool used for drawing and checking the Petri nets specifications.

Second, the advice checks whether all pre-conditions of the corresponding reconfiguration operation, specified in Z notation, are fulfilled. The translation of these pre-conditions into Java code makes use of a Java-based package that we developed for Z notation. This package contains classes representing the elements of the Z language such as operators, mathematical objects such as sets, relations, sequences, bags, etc. The developed methods are denoted `ZOperators` in the aspect template by (lines 11 and 14). Our aspect code generator parses the LaTeX file provided by the theorem prover Z/EVES and extracts the pre-conditions that should be translated. The constraints specified in the operation schema (line 11) and the constraints without quantification (line 14) specified in the system schema are evaluated first. These predicates are automatically translated to a call of one of the methods that are implemented in our Java package for Z notation, which corresponds to the used Z operator. Next, constraints that use quantification operators are evaluated by calling auxiliary methods generated for them (e.g., method `checkConstraintCi`, lines 18 and 30 in Listing 4.3). For example for the operator $\forall$, the auxiliary method verifies that all possible alternatives are satisfied.

The third phase consists of verifying the generated conditions that implemented the pre-conditions (lines 22–26). If one of the generated pre-conditions is not fulfilled the operation will not be executed (lines 26). Otherwise, the operation will be normally executed by calling the keyword proceed (line 23) and update the system states. For that, the aspect code generator emits a Java class that stores the current state of the system in terms of objects and relations between them. Each generated aspect implements a method `updateSystemState` (line 33), which updates the current state of the system if the reconfiguration operation can be executed. This method is called in the aspect after proceed (line 24).

```
1    //extract the pointcut from the mapping file
2     public pointcut PointCutName (parameters1): set of join points
3
4    //The around advice for controlling the reconfiguration operation
5     void around(parameters1): PointCutName(parameters1) {
6
7        // check if the Petri net transition is enabled
8        if (isTransitionEnabled (CurentMarking, respectiveTransistion)) { ...}
9
10       // evaluate the constraints defined in the operation schema
11       if (ZOperators(parameters2, SystemState)) { ... }
12
13       // evaluate constraints without quantification defined in the system schema
14       if (ZOperators(parameters3, SystemState)) { ... }
15
16       // call the auxiliary method for evaluating constraints with quantification
17        ...
18        if (checkConstraintCi()) { ... }
19        ...
20
21       // verify that all constraints hold and proceed if this is the case.
22        if (allConstraints) {
23            proceed(parameters1);
24            updateSystemState(parameters4);
25         }
26         else { ... }
27    }
28
29   // helper method for evaluating constraints with quantification
30   public boolean checkConstraintC_i() { ... }
31
32   // the method for updating the system state
33   void updateSystemState(parameters4) { ... }
34 }
```

Listing 4.3: Template of the generated AspectJ aspect

4.7 Overview of the Prototype

For proving the proposed concepts, we implemented an Eclipse plug-in that supports our approach. The main goal of our plug-in is to automatically generate AspectJ aspects from the Z and Petri net specifications of the architectural properties. Figure 4.3 presents a screenshot of our eclipse plug-in. Developers can invoke the plug-in the menu bar of Eclipse. In the following, we present the different functionalities that our plug-in provides.

Using the menu item *Load Petri net*, the developer loads the Petri-net specification that describes the execution order of the reconfiguration operations. This specification should be edited using the P3 tool and exported as an XML document. After that, the developer

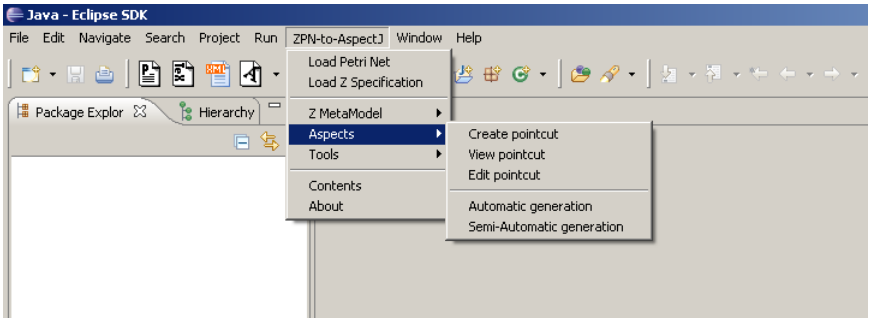

Figure 4.3: Screenshot of the ZPN-to-AspectJ plug-in

uses the menu item *Load Z specification* to load the Z specification that describes the architectural style and the reconfiguration operations. This specification should be edited using Z/EVES and exported as TEX file. After that, the plug-in automatically generates an XML document which represents the structure of the Z-PN specifications. This XML file is required to verify if the specifications (i.e., the model) are well-structured according to the defined patterns (i.e., the meta-model) as previously detailed in Section 4.5.5. This verification is performed by the menu item *Check*. In addition, the developer can visualize the generated XML file and the XDS file respectively through the menu item *Z2XML file* and *XSD file*. If the specification of architectural properties conforms to the implementation of the functional code, the developer can automatically generate the AspectJ aspects using the menu item *Automatic generation*. Otherwise, the developer should ensure first the mapping between the specification and the implementation by defining the corresponding pointcuts as an XML document. The menu items *Create pointcut*, *View pointcut*, and *Edit pointcut* allow respectively the creation, the visualization, and the editing of the pointcut. After that, the developer can generate the corresponding aspects according to the defined pointcuts using the menu item *Semi-Automatic generation*.

Our plug-in further allows developers to launch the Z/EVES and P3 tools to edit the specification of the architectural properties in Z-PN. It provides also some help for guiding the developer to better understand our proposed approach and to use our aspect generator.

4.8 Meidya: An Extension of the Seven-pro Approach

The programming model introduced by our approach is relatively complex, as we expect the user to have a good knowledge of the Z notation and Petri nets. We believe that the benefits of our approach make the increased programming-model complexity worthwhile.

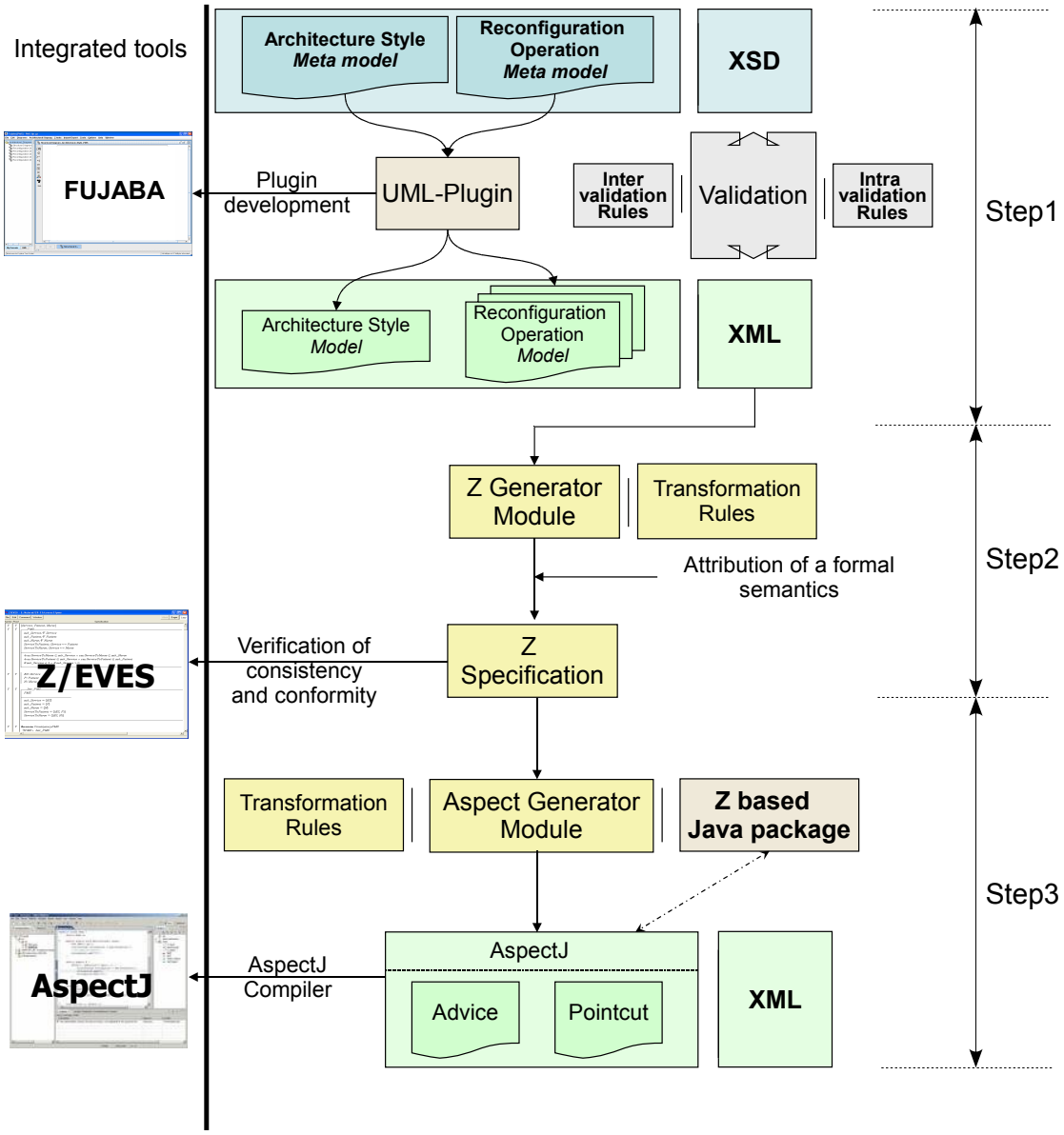

Figure 4.4: The Meidya approach

We deal with this problem by basing our approach on the work by Hadj Kacem et al. [101], who propose to graphically model dynamic software architectures based on the MDA modeling process. This approach provides modeling solutions to guide and assist

design activities. They define a new UML profile associated with two metamodels for describing dynamic software architectures. They propose a set of stereotypes and constraints for specifying architectural styles and reconfiguration operations as well as their pre- and post-conditions. This profile proposes also the use of the OCL language to specify the different types of architectural properties. The authors propose also automatic generation of Z specifications from UML models and the associated OCL constraints.

The SEVEN-pro approach complements the work proposed by Hadj Kacem et al. by formally verifying the generated Z specifications, and also by generating aspect code for verifying at runtime the architectural properties. The provided approach called MEIDYA [110, 111] covers the whole process for developing architectural properties of distributed applications, as shown in Figure 4.4. MEIDYA is based on three main steps:

In the first step, the designer specifies the dynamic software architecture using a UML profile. First, he defines the architectural style as well as the reconfiguration operations specified in terms of pre- and post-conditions. Second, the system maps automatically UML models towards the XML language. This function ensures that each generated XML descriptions is valid according to the appropriate XML schema defining the proposed metamodel (as previously defined in Section 4.5.5).

In the second step, the system automatically translates the graphic UML models to Z specifications. The designer can interact with a theorem prover, such as Z/EVES, to verify the consistency of the architectural style, on the one hand, and the preservation of the architectural invariants during the architectural evolution, on the other hand.

In the third step, the programmer uses our aspect generator to automatically produces AspectJ aspects from Z specifications. The generated aspects will be integrated in a modular way in the functional application code in order to enforce the architectural invariants at runtime.

A software environment supporting these features has been developed and integrated as a plug-in in the open source tool FUJABA [110].

Using the MEIDYA approach, the designer models the software architecture of his distributed applications using UML and OCL: No knowledge about the Z notation is required. MEIDYA defines a high level language enabling specification of generic and reusable model. In addition, this approach seeks to take advantage of the standard visual notations provided by UML2.0 and the associated OCL language. It bridges the gap between the steps of the development process by automatically generating Z specifications and AspectJ code.

4.9 Related Work

In this section, we present an overview of related work on the specification, verification, and enforcement of architectural properties.

4.9.1 Specifying architectural properties

There have been many works on modeling dynamic software architectures. The proposed approaches can be classified into three classes.

The first class uses Architecture Description Languages (ADL) to model software architectures. There are two main limitations: First, some of these ADLs do not provide any mechanisms for formally verifying the architectural properties such as the absence of contradiction between constraints. Thanks to the theorem prover Z/EVES, the SEVEN-pro approach can verify the consistency and other application-specific properties. Second, according to [100], the majority of ADLs are unsuitable for defining complex properties and describing pre- and post-conditions, and coordination constraints of architecture reconfiguration operations. In our approach and based on logic predicate and set theory of Z notation as well as Petri nets, the designer can easily specify very interesting properties.

The second class uses UML and OCL for modeling software architectures. The expressive power of UML is limited for expressing dynamic software architectures. The UML diagrams cannot specify the reconfiguration operations and their pre- and post- conditions. The common solution to solve this limitation is to extend UML to support concepts for modeling dynamic architectures, such as [101, 168]. These approaches lack formal foundations for verifying the consistency or other specific properties. An automatic translation of UML models to formal languages, as proposed in MEIDYA can solve this limitation.

The third class of approaches uses formal methods. We classify them according to the techniques that they use: (1) approaches based on logic and (2) approaches based on process. In the following, we focus on works that use Z notation and Petri nets.

Some researchers propose using *logic-based* approaches such as first-order logic [65], Z [2], and temporal logic [4]. First-order logic can cover only the architecture invariants and pre- and post-conditions of architecture-reconfiguration operations, whereas temporal logic can express some coordination properties and temporal constrains at a very high level.

Abowd et al. [2] propose using Z to analyze the architecture styles and the relation between them. However, this work does not support the modeling of dynamic software architectures and their constraints. Based on work by Loulou et al. [133], we combine Z notation and Petri nets to specify dynamic software architectures, associated with their architectural invariants, pre- and post-conditions, as well as the coordination constraints. Thus, we can formally verify several types of properties such as the consistency of our specification, the maintaining of the invariants when the architecture evolves, and the absence of deadlock in Petri nets specifications.

Other approaches use *process-based* formalisms such as CSP [92], π-calculus [155], graph grammars [141], and Petri nets [73, 90].

Le Métayer [141] uses a graph grammar, based on a mathematical model, to formally specify dynamic software architectures. He defines the architecture in terms of graphs and

the architecture style using graph grammars. Constraints on the evolution of the architecture are specified by means of graph rewriting rules. In addition, an algorithm is defined to verify the consistency of the rewriting rules to prove that the architectural constraints are preserved after a reconfiguration. Compared to the SEVEN-pro approach, this type of grammars does not support logical properties such as reasoning about the number of object (component) instances and logical conditions such as absence of a communication link between two objects. In addition, the designer can verify also other specific properties using the Z/EVES theorem prover.

SAM [90] (Software Architecture Model) is a general software architecture development framework based on Petri nets and temporal logic. Petri nets are used to model the structure and the behavior of components and connectors, while temporal logic is used to specify the required temporal properties of the given architecture. This work is related to ours as it uses two formalisms to specify the architectural properties. However, the SAM does not support dynamic software architectures and corresponding pre- and post-conditions of reconfigurations. In our approach, we use the Z notation and specially Z operation schema, which is well suited to define the reconfiguration operations and their constraints. In addition, we focus only on the architectural level, we are not interested in the behavior of components.

In [73], the authors propose a dynamic software architecture model based on π-net, a formal language that combines object-oriented Petri nets with π-calculus. The proposed model takes into account the structure and dynamic aspects of software architecture. Compared to ours, this approach does not support the specification of architectural properties. No verification mechanism has been proposed. In addition, as other process-based formalism, the π-calculus is a very high-level language and cannot be easily translated to code. In our approach, we implement a Java package for helping the translation of Z predicates to code and we generate also a code that enforces the Petri nets specifications.

Compared to our approach, most of the approaches mentioned above do not support code generation from the specifications of software architectural properties. Moreover, they do not cover all architectural properties mentioned in Section 4.2. In the next section, we introduce some approaches that address the enforcement of architectural properties.

4.9.2 Enforcing architectural properties

ArchJava [8] is an extension to Java that seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to the architectural constraints. It extends a practical implementation language (Java) to incorporate architectural features and enforce communication integrity. Although the approach allows developers to enforce architecture constraints, it mainly focuses on communication integrity. ArchJava does not

support other types of architectural reasoning, such as reasoning about coordination protocols and architectural styles.

DiscoTect [195] is a tool that allows to determine the architecture of a system using dynamic analysis for multiple architectural styles and multiple systems. The implementation level events are mapped into architectural operations using a modified version of a state machine. This tool uses online monitors for system observation to guarantee the consistency of the implementation with its architectural design. DiscoTect does not support the specification and the enforcement of dynamic software architectures.

SonarJ [91] is a commercial Eclipse plug-in for enforcing architectural constraints in Java programs. This plug-in allows an efficient enforcement of architectural constraints, but it does not support coordination constraints and many constraints that can be defined with Z notation. The previously mentioned tools do not have a formal basis and do not ensure any modularity for monitoring and controlling software architecture evolution. Few approaches address the use of aspect-oriented techniques for specifying and enforcing architectural properties.

Some ADLs have been extended by aspect paradigms such as [25, 57, 132] to specify and enforce architectural properties in a modular way. For example, Loukil et al. propose AO4AADL, an aspect-oriented extension for AADL. This new language allows specifying non-functional properties and specially architectural properties. The authors propose also an AO4AADL compiler that generates AspectJ aspects, which verify at runtime if the running system conforms the AAO4ADL specifications. In general, these extended ADLs allow to model and enforce architectural properties in a modular way. Compared to our approach, they do not present any formal foundations to verify architectural properties.

In [135], the authors present an approach for enforcing predefined architectural styles (e.g. client/server, publish/subscribe, etc.) and their constraints on top of the Prism-MW, an architecture middleware platform. This approach allows developers to modify the architectural style without impacting the rest of the system. For each architectural style, an AspectJ aspect is defined. In [135], only the structure of the architectural style is supported without considering the architectural constraints defined in Section 4.2.

4.10 Conclusion and Limitations

By considering architectural properties on the specification level one can more easily manage the reconfiguration of the dynamic architecture and detect errors and faults earlier in the development process. In addition, by automatically generating verification code from the specified properties, one can define more reliable distributed applications. The SEVEN-pro approach for architectural properties guarantees these previous advantages.

This chapter presented an approach for specifying architectural properties of distributed

applications based on dynamic architecture. The combination of Z and Petri nets allows us to specify most important architectural properties. Z notation is used to specify invariants, pre- and post-conditions, while we use Petri nets to model coordination protocols. We also presented a verification phase that allows developers to verify the consistency of architectural properties and the preservation of the system after a reconfiguration of the architecture.

This chapter also presented the aspect based enforcement of the specified properties. AspectJ code is generated from the Z and Petri-net specifications. The generated aspects control the evolution of the architecture at runtime. The aspects prohibit the execution of the reconfiguration operations if their corresponding pre-conditions are not satisfied.

Our approach has some limitations. First, as we use Petri nets, we support only applications with finite states. The use of linear-temporal logic, extended with Z notation, can resolve this problem. Second, the generated aspects are complex and additional code is needed to explicitly keep track of the system state. This is because of the limited power of querying capabilities of AspectJ. More expressive pointcut languages such as Alpha [156] support a more declarative quantification over execution history and object heap. By using them, we hope to achieve a more direct mapping of formal specifications to aspect code. Such pointcut languages are also less fragile with respect to syntactic changes.

In the next chapter, we will detail the application of the SEVEN-pro approach for implementing access control policies. We use *TemporalZ* as a specification language for specifying separation of duties and delegations policies, and we use ALPHA as an expressive aspect language for runtime monitoring the specified policies.

Chapter 5
Specifying and Enforcing Access Control Policies

5.1 Introduction

This chapter shows how we applied SEVEN-pro approach for implementing secure applications. The contributions presented in this chapter were published in [107–109].

We address the specification and enforcement of security policies that are enforceable by runtime monitoring as characterized by Schneider [173]. More specifically, we propose an approach to the runtime verification of separation of duties and delegation policies on top of the role-based access-control policy. These policies allow defining user roles and controlling user permissions in a multi-user system. They ensure that only authorized users can call and execute certain functions of the system. For example, separation-of-duties policies are access-control policies that allow defining constraints on the assignment of users to the critical activities of a business process to prevent frauds and errors.

The formal specification phase of the SEVEN-pro approach is covered by a new formal specification language called *TemporalZ*, which combines Z and linear temporal logic. We extend this language by domain specific predicates for specifying access control policies. Based on this language, we verify that there is no contradiction between security predicates. In the runtime enforcement phase, we propose to encode the enforcement logic as aspects in the aspect-oriented language ALPHA, which has built-in means for reasoning about an expressive model of the program execution, including the execution history and the object store by using logic programming. We also extend ALPHA by a new library to easily enforce the access control predicates as aspects.

The remainder of this chapter is organized as follows. Section 5.2 introduces role based access control, separation of duties, and delegation policies. In Section 5.3, we present the case study, a loan-approval process, used throughout this chapter. An overview of the approach is given in Section 5.4. Section 5.5 introduces *TemporalZ* and explains the formal specification of separation of duties and delegation policies. Section 5.6 explains how formal

specifications are automatically mapped to ALPHA aspects. Section 5.7 reports on related work. Section 5.8 concludes this chapter and discusses areas of future work.

5.2 Access Control Policies

As defined in [170], *access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied*. Several access-control policies have been proposed in literature such as DAC (Discretionary Access Control), MAC (Mandatory Access Control), and RBAC (Role-Based Access Control). In this work, we focused on the formal specification and the enforcement of RBAC policies, extended by several types of separation of duties and delegation policies.

5.2.1 Role-Based Access Control

Role-Based Access Control (RBAC) [69] is an authorization mechanism in which access decisions are based on the user roles in the organization (Figure 5.1, taken from [69]). The permissions to execute a set of operations are grouped in roles and users are assigned to one or more roles. A role can contain many users and a user can be member of many roles. Also, a role can have many permissions, and a permission can be assigned to many roles. For example, the security administrator of the banking software may define the following roles for LAP: *Teller*, *FinancialClerk*, *Supervisor*, *Manager*, and *Customer*. Each role has operations associated with it, e.g., the role *Teller* includes permissions for the operations *EnterApplicationData*, *CheckExternalRating* and *TransferMoney*.

The session and role hierarchies concepts have been proposed to enrich the basic RBAC policy. The session concept ensures the mapping between a user and an activated subset of roles that are assigned to the user. The role hierarchy allows to structure the roles in a large organization and it is based on inheritance relationships between roles. The role hierarchy enables a senior role to perform the permission of a junior role.

In this work, we focus on the separation-of-duties [79] and delegation policies [54] on top of RBAC.

5.2.2 Separation of Duties

Separation-of-duty (SoD) policies reduce the risk of frauds and errors in business processes by a fine-grained control over the privileges for workflow tasks. Two main categories are distinguished: static and dynamic separation of duties [79].

- **Static SoD (SSoD)** This class of separation of duty specifies that two mutually exclusive roles must never be assigned to the same user simultaneously, e.g., a bank

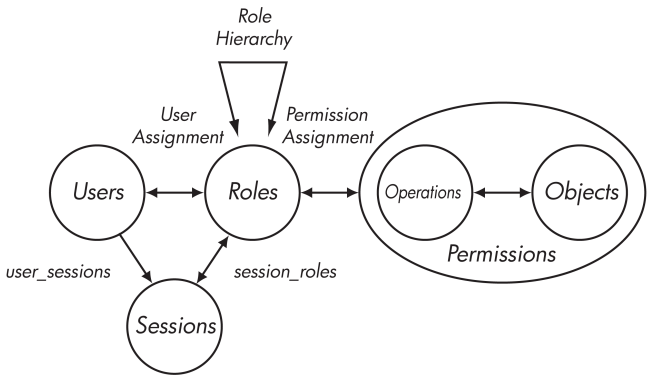

Figure 5.1: Role-based access control model

employee cannot be assigned, at the same time, the roles *Supervisor* and *Manager*.

- Dynamic SoD (DSoD) This class of policies takes the dynamics of the system execution into account for managing roles and role membership. Four variations of dynamic SoD are distinguished in the literature:

Simple Dynamic SoD (SDSoD) two mutually exclusive roles must never be activated by a user at the same time, e.g., a bank employee can be statically assigned both roles *Teller* and *FinacialClerk*, but cannot activate them simultaneously.

Object-Based SoD (ObjDSoD) a user can activate two exclusive roles at the same time, but he cannot act upon the same object via both roles. E.g., if a user in LAP can be *Supervisor* and *Teller*, then he would be able to execute the operations of the role *Teller* and verify his own work using the *Supervisor* role. For example, an ObjDSoD constraint in the loan application example may state that the user can activate the role *Supervisor* and *Teller* at the same time as long as he does not act on the same loan application object.

Operational SoD (OpDSoD) a user can activate two mutually exclusive roles at the same time, but cannot have all required authorizations to execute all tasks in a workflow process. For instance, the critical sub-processes (e.g., *enterApplicationData*, *verifyRating*, *verifyTransfer*) cannot be executed by the same employee.

Operational Object-Based SoD (OpObjDSoD) combines operational and object-based SoD; a user can activate two exclusive roles at the same time and can have the authorizations to execute all tasks in a workflow process, as long as these tasks do not act upon the same object.

5.2.3 Delegation

Delegation [54] is a further extension of the basic RBAC that allows users to assign all or a part of their permissions to other users. For example, when a manager takes a vacation, he may assign some of his permissions to his assistants so that they can carry out his tasks. Also, many projects require the collaboration of different people, which means that delegating certain permissions is necessary to allow such collaborations. In the rest of this chapter, we call the user who performs the delegation task "delegator", and the user who receives the access right "delegatee". Two types of delegation can be distinguished:

- **User-to-User** (U2U for short) the delegator can delegate his rights at the same time to a single user or a set of users.
- **User-to-Role** (U2R for short) the delegator delegates his rights to a role (i.e. all users assigned to this role).

Several delegation characteristics have been defined on top of RBAC policy:

- **The totality characteristic** leads to two types: partial and total delegation. In the first one, called also *permission delegation*, the delegator delegates a part of his individual access permissions. In the second, called *role delegation*, the delegator delegates his role, which allows the delegatee to act in this role.
- **The monotonicity characteristic** refers to whether roles/permissions are kept after delegation or not. Two types of delegation exist based on this characteristic: *grant* and *transfer* delegation. In the grant delegation, the delegator maintains the permissions that he delegated after the delegation step. In the transfer delegation, the delegator gives up the delegated permissions after the delegation step.
- **The level of delegation** defines whether the delegated role/permission can be further delegated. The single-step delegation means that the delegated right cannot be delegated another time in the future. The multi-step delegation means that the delegated right can be delegated many times.

5.3 Case Study: A Loan Approval Process

To illustrate our approach, we consider a simple loan approval process (LAP) in a bank (inspired from [172]). We first describe the different steps in that process in an informal manner and then define the associated user roles such as teller, financial clerk, supervisor, manager, and customer.

1. **Initial loan application** (Enter application data): In the first step, a *teller* interacts with the *customer* to identify the loan program for which he qualifies and can apply. The *teller* also verifies the financial situation of the customer (e.g., account type, income, liabilities, assets, etc.) and checks the documents provided together with the application. If the application is complete and accurate the loan approval process will be started.

2. **Rating process** (Check internal rating – Check external rating – Verify rating): In this step, two types of rating are performed based on the application and the documents provided with it, the credit history of the customer, and other external information. The first step, which is about internal rating, is done by one or more financial clerks *FinancialClerk* to verify whether the customer financial situation is acceptable for the loan program he is applying for. The second rating is the external rating, in which a bank employee in the role *Teller* interacts with some external agency to check whether the customer has financial problems with other banks or companies such as unpaid bills, etc. In the third step, an employee in the role *Supervisor* verifies whether the internal and external rating steps were done appropriately in order to avoid errors and prevent the risk of fraud.

3. **Bank decision** (Take decision): Based on the results of the internal and external ratings, and the level of risk defined by the *Supervisor*, a decision is made on the loan application. If the loan application is approved, a supervisor gives the order to create the contract and proceed with the signature steps; otherwise the process will be canceled.

4. **Contract signature** (Sign customer contract – Sign manager contract): After the loan application has been approved, a contract document is prepared and sent to the *manager*, who is the bank legal representative and to the *customer* for signature.

5. **Loan closing** (Transfer money – Verify transfer): In this step, the *teller* who entered the loan application data transfers the approved amount of money to the customer. To eliminate any possibility of error an employee, belonging to the *supervisor* role, verifies this money transfer step.

In the rest of this chapter, we will use this case study to explain the formal specification and the runtime enforcement of separation of duties and delegation policies.

5.4 The Approach in a Nutshell

We applied the SEVEN-pro approach to access control policies. The instantiation of the approach in this context gave rise to a three-steps approach for the development of secure

applications. We show this approach in Figure 5.2. The approach supports the specification and enforcement of role-based access control, separation of duties and delegation policies.

The security designer uses *TemporalZ* to formally specify the RBAC policy and the different policies and constraints on top of it. Then, he verifies the consistency of the formal specification by using the tool Z-EVES for theorem proving. To facilitate this specification task, we proposed some specification patterns for structuring the whole specification and for formally specifying separation of duty properties.

After that, the application developer implements the functional code of the application using Java. The functional code does not contain any logic for authorization, i.e., if the application implements a certain workflow process then any user will be able to execute any task in the workflow. In addition, the implementation of the application should conform to the specification, i.e., all classes, methods, and fields that are mentioned in the specification should be implemented.

Once both the application logic is implemented and the security policies are specified, the developer can use our aspect generator tool to generate the security module of the application. The tool translates the formally specified authorization policies in *TemporalZ* to separate ALPHA aspects. At runtime, these aspects will be triggered at appropriate points during the execution of the application and the respective advice will enforce the specified security policies.

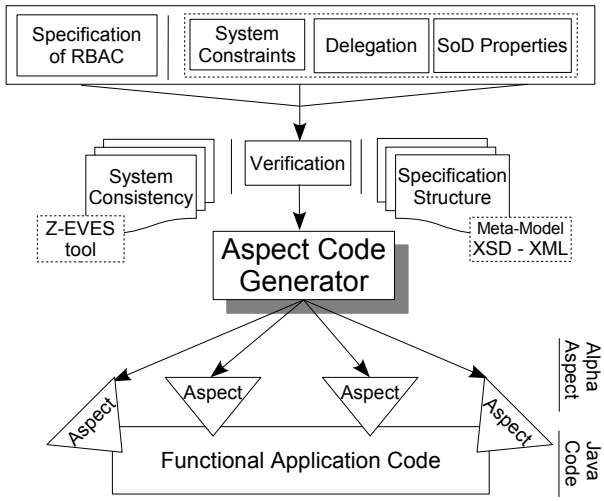

Figure 5.2: Specifying and enforcing access control policies

5.5 Specification of Access Control Policies

In this section, we detail the formal specification and verification of the separation-of-duties properties and delegation policies on the top of role-based access control policy. We used *TemporalZ* as a domain specific specification language for expressing access control policies.

5.5.1 Formal language: *TemporalZ*

TemporalZ [167] integrates linear temporal logic into the Z framework. We formally define the syntax and the semantics of *TemporalZ*. First, we define the syntax of the temporal formulas, which combine temporal operators and security predicates that are required to specify temporal properties. As shown in Figure 5.3, a *TemporalZ* formula is either (a) a *domain specific predicate* (DSPredicate), or (b) constructed from other formulas using the *temporal operators*: $\Box$(always), $\Diamond$(eventually), $\bigcirc$(next), $\boxminus$(up till now), $\diamondminus$(previously), $\ominus$(atlast), $\mathcal{U}$ (until), and $\mathcal{S}$ (since).

$$
\begin{aligned}
&\mathit{Formula} ::= \mathit{DSPredicate} \\
&\quad | \ \Box\langle\langle \mathit{Formula} \rangle\rangle \mid \Diamond\langle\langle \mathit{Formula} \rangle\rangle \mid \bigcirc\langle\langle \mathit{Formula} \rangle\rangle \\
&\quad | \ \boxminus\langle\langle \mathit{Formula} \rangle\rangle \mid \diamondminus\langle\langle \mathit{Formula} \rangle\rangle \mid \ominus\langle\langle \mathit{Formula} \rangle\rangle \\
&\quad | \ \mathcal{U}\ \langle\langle \mathit{Formula} \times \mathit{Formula} \rangle\rangle \\
&\quad | \ \mathcal{S}\ \langle\langle \mathit{Formula} \times \mathit{Formula} \rangle\rangle
\end{aligned}
$$

Figure 5.3: Syntax of *TemporalZ*

The semantics of these formulas is defined as an evaluation according to a temporal model presented below. This model is specified as a function which assigns a time t to the set of specific predicates that hold at t. We formalize this by writing *Time* for a natural number denoting the current time and *Model* for the function that maps each point in time to a finite set of *formulas*.

$$
\begin{aligned}
&\mathit{Time} == x : \mathbb{N} \\
&\mathit{Model} == \mathit{Time} \rightarrow \mathbb{F}\, \mathit{Formula}
\end{aligned}
$$

We define three axiomatic functions that allow the evaluation of temporal formulas at different abstraction levels. *EvalMT* interprets each formula at a given point in time t, using the temporal model m. *EvalMT* evaluates domain-specific predicates without temporal operators to *True* (E1), if, according to m, the predicate holds at t. For composed formulas, a variant of *EvalMT* is defined for each temporal operator; for brevity, only the evaluation of formulas composed with *eventually* $\Diamond$ (E2) is shown below.

$$
\begin{array}{|l}
EvalMT\_ : \mathbb{P}(Formula \times Model \times Time) \\
\hline
\forall\, dsp : DSPredicate; m : Model; t : Time \\
\quad \bullet\ EvalMT(dsp, m, t) \Leftrightarrow dsp \in m(t)
\end{array}
\qquad [\mathsf{E1}]
$$

$$
\begin{array}{|l}
\forall\, f : Formula; m : Model; t : Time \\
\quad \bullet\ EvalMT((\Diamond(f)), m, t) \Leftrightarrow \\
(\exists\, t1 : Time \mid t1 \geq t \bullet EvalMT(f, m, t1) \\
\ldots
\end{array}
\qquad [\mathsf{E2}]
$$

Then, we generalize our evaluation by abstracting away the time parameter and obtain a new function that evaluates temporal formulas according to the model as shown below.

$$
\begin{array}{|l}
EvalM\_ : \mathbb{P}(Formula \times Model) \\
\hline
\forall\, f : Formula; m : Model \bullet EvalM(f, m) \\
\qquad \Leftrightarrow (\forall\, t : Time \bullet EvalMT(f, m, t))
\end{array}
\qquad [\mathsf{E3}]
$$

After that, we define a second generalization by abstracting away the model parameter and we obtain the following evaluation function.

$$
\begin{array}{|l}
Eval\_ : \mathbb{P}\, Formula \\
\hline
\forall\, f : Formula \bullet Eval(f) \Leftrightarrow (\forall\, m : Model \bullet EvalM(f, m))
\end{array}
\qquad [\mathsf{E4}]
$$

Finally, we define equivalence relationships between each temporal operator and its evaluation definition as shown below for the temporal operators *Eventually* ([E5]) and *Until*([E6]). This logical equivalence allows us to specify the temporal constraints in a more convenient way and facilitates their translation to code.

$$
\begin{array}{|ll}
\forall\, f : Formula \bullet Eval(\Diamond f) \Leftrightarrow \Diamond f & [\mathsf{E5}] \\
\forall\, f, g : Formula \bullet Eval(\mathcal{U}\ (f, g)) \Leftrightarrow f\ \mathcal{U}\ g & [\mathsf{E6}] \\
\ldots &
\end{array}
$$

5.5.2 *TemporalZ* for Access Control policies

We use *TemporalZ* for the formal specification of the RBAC policy and stateful access control policies such as the separation-of-duty and delegation policies. For that purpose, we extend the syntactic and semantic definition from the previous Section with domain-specific security predicates. The syntactic definition of these predicates is shown in Figure 5.4 whereas their semantic definition is shown in Figure 5.5, using the evaluation function *EvalMT* ([E2]).

The predicate *include(u,r)* (Figure 5.5, E7) tells whether the user u is one of the users that exist in the current *RBAC* system, is assigned to the role r.

$$
\begin{aligned}
DSPredicate ::= & \; include\langle\langle USER \times ROLE \rangle\rangle \mid active\langle\langle USER \times ROLE \rangle\rangle \\
& \mid execOp\langle\langle USER \times ROLE \times OPERATION \rangle\rangle \\
& \mid execOpObj\langle\langle USER \times ROLE \times OPERATION \times OBJECT \rangle\rangle \\
& \mid execSeqOp\langle\langle USER \times \text{seq } OPERATION \rangle\rangle \\
& \mid execSeqOpObj\langle\langle USER \times \text{seq } OPERATION \times OBJECT \rangle\rangle
\end{aligned}
$$

Figure 5.4: Syntax of security predicates

The predicate *active(u,r)* (Figure 5.5, E8), verifies that the user u has activated the role r, i.e., the user should be belonging already to the role r (defined using the predicate *include*) and should have connected to that role within a session s (defined using the relation *UserSession*). In the implementation phase, the activation of a session in a specific role is implemented by the fact that the user executes for the first time an operation belonging to this role.

Figure 5.5, E9, defines the semantics of the domain specific predicate *execOp(u, r, op)*. This predicate verifies whether the user u has already the access to execute the operation *op*. The user u should be already a member of the role r and can active it (defined by the predicate *include*). This role r should contain the operation *op* (defined using the relation *RoleOperation*).

The predicate *execOpObj(u, r, op, obj)* (Figure 5.5, E10) refines the predicate *execOp* by checking whether the user u can execute the operation op with the object *obj* as parameter. In addition to the constraints defined for the predicate *execOp*, the the object *obj* should be specified as an input parameter for the operation *op* (defined using the relation *OperationObject*).

The predicate *execSeqOp* (Figure 5.5, E11) introduces the concept of an ordered operation sequence. The predicate *execSeqOp(u,sop)* tells whether the user u can execute the sequence of operations *sop* in the order defined by the sequence. The user u, belonging to any role, should first have the access to execute all operations of the sequence *sop* (using the predicate *execOp*). Second the execution of these operations should be conforms to the mentioned order in the sequence, using future and past temporal operators. i.e., the operation with the order *(i)* denoted by *Sop(i)* cannot be executed only after the execution of the operation with the order *(i-1)* denoted by *Sop(i-1)*.

The predicate *execSeqOpObj(u,sop,obj)* (Figure 5.5, E12) refines the predicate *execSeqOp* to express that all operations are executed on the same object *obj*. Instead of using the predicate execOp to verify if the user has the access to execute the sequence operations, *execOpObj* is used to ensure in addition that this operation can be executed on the object *obj*.

E7:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; r : ROLE \\ \bullet\ EvalMT((include(u, r)), m, t) \Leftrightarrow \\ (\exists\, system : RBAC \bullet u \in system.users \\ \land\ r \in system.roles \land u \in system.RoleUser(\!|\ \{r\}\ |\!))\end{array}$$

E8:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; r : ROLE \bullet \\ \exists\, s : SESSION \bullet EvalMT((active(u, r)), m, t) \Leftrightarrow \\ (\forall\, system : RBAC \bullet EvalMT((include(u, r)), m, t) \\ \land\ s \in system.sessions \land s \in system.UserSession(\!|\ \{u\}\ |\!) \\ \land\ s \in system.RoleSession(\!|\ \{r\}\ |\!))\end{array}$$

E9:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; r : ROLE; \\ op : OPERATION \bullet EvalMT((execOp(u, r, op)), m, t) \Leftrightarrow \\ (\forall\, system : RBAC \bullet EvalMT((active(u, r)), m, t) \\ \land\ op \in system.operations \land op \in system.RoleOperation(\!|\ \{r\}\ |\!))\end{array}$$

E10:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; r : ROLE; \\ op : OPERATION; obj : OBJECT \\ \bullet\ EvalMT((execOpObj(u, r, op, obj)), m, t) \\ \Leftrightarrow (\forall\, system : RBAC \bullet EvalMT((execOp(u, r, op)), m, t) \\ \land\ obj \in system.objects \land obj \in system.OperationObject(\!|\ \{op\}\ |\!))\end{array}$$

E11:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; Sop : \mathrm{seq}\ OPERATION \\ \bullet\ EvalMT((execSeqOp(u, Sop)), m, t) \\ \Leftrightarrow (\forall\, i : \mathbb{N} \mid 1 \leq i \leq \#Sop \\ \bullet\ (\exists\, r : ROLE \bullet EvalMT((\Diamond(execOp(u, r, (Sop(i))))), m, t))) \\ \land\ (\forall\, i : \mathbb{N} \mid 1 < i \leq \#Sop \\ \bullet\ ((\exists\, r1 : ROLE \bullet EvalMT((execOp(u, r1, (Sop(i)))), m, t)) \\ \Rightarrow (\exists\, r2 : ROLE \bullet EvalMT((\Diamond\!\!-(execOp(u, r2, (Sop(i-1))))), m, t))))\end{array}$$

E12:

$$\begin{array}{l}\forall\, m : Model; t : Time \bullet \forall\, u : USER; Sop : \mathrm{seq}\ OPERATION; \\ obj : OBJECT \bullet EvalMT((execSeqOpObj(u, Sop, obj)), m, t) \\ \Leftrightarrow (\forall\, i : \mathbb{N} \mid 1 \leq i \leq \#Sop \\ \bullet\ (\exists\, r : ROLE \bullet EvalMT((\Diamond(execOpObj(u, r, (Sop(i)), obj))), m, t))) \\ \land\ (\forall\, i : \mathbb{N} \mid 1 < i \leq \#Sop \\ \bullet\ ((\exists\, r1 : ROLE \bullet EvalMT((execOpObj(u, r1, (Sop(i)), obj)), m, t)) \\ \Rightarrow (\exists\, r2 : ROLE \bullet EvalMT((\Diamond\!\!-(execOpObj(u, r2, (Sop(i-1)), obj))), m, t))))\end{array}$$

Figure 5.5: Semantics of security predicates

The proposed specification language *TemporalZ* is well suited to specify access control policies for several reasons: Set theoretic and predicate logic foundations of Z directly support the expression of RBAC concepts such as roles, rights, and related constraints. Moreover, the use of LTL temporal operators in *TemporalZ* allows expressing temporal constraints on system behavior in a declarative way. Further, complementing the general-purpose elements of Z and LTL with domain-specific predicates increases the level of abstraction of the specifications and makes them more declarative. Finally, tools for verification and theorem proving such as Z-EVES [139] can be used to verify the consistency of the formally specified security policies.

5.5.3 Specification of the RBAC

RBAC

$$
\begin{array}{l}
roles : \mathbb{F}\, ROLE; users : \mathbb{F}\, USER; sessions : \mathbb{P}\, SESSION \\
operations : \mathbb{F}\, OPERATION; objects : \mathbb{F}\, OBJECT \\
RoleOperation : ROLE \leftrightarrow OPERATION \qquad [D1] \\
RoleUser : ROLE \leftrightarrow USER \qquad [D2] \\
RoleSession : ROLE \leftrightarrow SESSION \qquad [D3] \\
UserSession : USER \leftrightarrow SESSION \qquad [D4] \\
OperationObject : OPERATION \leftrightarrow OBJECT \qquad [D5] \\
\hline
\mathrm{dom}\, RoleOperation \subseteq roles \qquad [C1] \\
\mathrm{ran}\, RoleOperation \subseteq operations \qquad [C2] \\
\mathrm{dom}\, RoleUser \subseteq roles \\
\mathrm{ran}\, RoleUser \subseteq users \\
\ldots \\
\forall\, u : users; r : roles \bullet \exists\, s : sessions \bullet (r, u) \in RoleUser \\
\qquad \Leftrightarrow (u, s) \in UserSession \land (r, s) \in RoleSession \qquad [C3] \\
\ldots
\end{array}
$$

Figure 5.6: Specification of the RBAC model

The specification of the basic RBAC system in *TemporalZ* is shown in Figure 5.6. The upper part defines sets of roles, users, operations, objects, and sessions, as well as relations between them. Each role has a set of operations associated with it, defined by *RoleOperation* [D1]. A user should have the role, as defined by the relation *RoleUser* [D2], to be allowed to execute the operations of that role. A session is a mapping of a user to a subset of his roles as defined by ([D3] and [D4]). The relation *OperationObject* [D5] maps operations to objects they operate on. The lower part of Figure 5.6 defines constraints on RBAC model

in terms of first-order predicates: For example, [C1] and [C2] define the ranges and domains of the relations defined in the upper part. [C3] specifies that a user establishes a session during which the user activate some roles that he is assigned.

In *SystemConstraints* schema, the designer can specify the constraints that depends on the application. Figure 5.7 defines constraints on LAP that restrict the number of (a) roles [C4], (b) users assigned to the role *Teller* [C5], and (c) operations belonging to a role [C6]. We can also define other types of constraints, e.g., an operation cannot be included in more than one role [C7].

SystemConstraints

$RBAC$

$\#roles < 10$	[C4]
$\#(RoleUser(\!\vert \{Teller\} \vert\!)) \leq 5$	[C5]
$\forall r : roles \bullet \#(RoleOperation(\!\vert \{r\} \vert\!)) \leq 7$	[C6]
$\forall p : operations; r1, r2 : roles \mid r1 \neq r2$	
$\quad \bullet\ p \notin RoleOperation(r1) \lor p \notin RoleOperation(r2)$	[C7]

Figure 5.7: Specification of the system constraints

Each administrative operation, e.g., assigning a role to a user, is specified by an operation schema, which defines its input parameters and its pre- and post-conditions. For example, in LAP, we can specify that a new user cannot be a member in any role or any special role, as shown in Figure 5.8.

assignUserToRole

$\Delta RBAC$

$u? : USER; r? : ROLE$

$\forall r : role \bullet (r, u?) \notin RoleUser$	[C8]
$users' = users \cup \{u?\}$	
$RoleUser' = RoleUser \cup \{r?, u?\}$	
$\ldots$	

Figure 5.8: Specification of the prerequisite constraints

5.5.4 Specification of SoD Properties

We proposed some specification patterns using *TemporalZ* to facilitate and to structure the specification of SoD properties. Similar properties (having the same SoD type) are specified in a Z schema which refines the RBAC specification. The declarative part of this schema contains the name of the RBAC specification schema while the predicative part contains the specification of the SoD properties. We propose a pattern for each SoD type.

The specification pattern for static SoD is shown in the following Z schema. This pattern uses the predicate *include* defined in (Figure 5.5, E7) to state that a user *USER* can be a member only of one of two exclusive roles ($Role_1$ and $Role_2$) at the same time. The predicate *ExclusiveRole* is used only in static SoD to ease the translation of the SoD property to ALPHA code and to avoid the conflict between SoD and delegation policies.

Static_SoD_Pattern

$$RBAC\_model\_schema$$

$$\forall Role_1, Role_2 : ROLE \bullet ExclusiveRole(Role_1, Role_2) \Leftrightarrow \forall u : USER \bullet \neg\, include(u, Role_1) \vee \neg\, include(u, Role_2)$$

For example, the Z schema above specifies that an employee in LAP (cf. Section 5.3) cannot be a *Manager* and a *Supervisor* at the same time.

StaticSoD

$$SystemConstraints$$

$$ExclusiveRole(Supervisor, Manager) \Leftrightarrow \forall u : USER \bullet \neg\, include(u, Supervisor) \vee \neg\, include(u, Manager)$$

The specification pattern *Dynamic_SoD_Pattern* uses the predicate *active* (Figure 5.5, E8) to state that an employee can be a member in two roles $Role_1$ and $Role_2$, but not simultaneously.

Dynamic_SoD_Pattern

$$RBAC\_model\_schema$$

$$\forall u : USER; Role_1, Role_2 : ROLE \bullet \Box(\neg\, (active(u, Role_1) \wedge active(u, Role_2)))$$

We applied dynamic separation of duty to the *Teller* and the *FinancialClerk* roles: a teller (resp. financial clerk) employee cannot execute any operations belonging to the financial clerk (resp. teller) role.

DynamicSoD

$$SystemConstraints$$

$$\forall u : USER \bullet \Box(\neg\, (active(u, teller) \wedge active(u, financialClerk)))$$

Using the predicates *execOp* and *execOpObj* (Figure 5.5, E9 and E10), the specification *ObjectBasedSoD* states that a user who has executed the $Operation_1$ in the $Role_1$ can never in the future execute the $Operation_{21}$ and the $Operation_{22}$ in the $Role_2$ on the same object *obj*.

Object_Based_SoD_Pattern

RBAC_model_schema

$$\forall u : USER; obj : OBJECT; Role_1, Role_2 : ROLE \bullet$$
$$\Box((execOpObj(u, Role_1, Operation_1, obj))$$
$$\Rightarrow (\neg (\Diamond((execOpObj(u, Role_2, Operation_{21}, obj))$$
$$\wedge (execOpObj(u, Role_2, Operation_{22}, obj))))))$$

The following *TemporalZ* schema presents an example of the specification of object-based SoD on the LAP case study. An employee can belong to the roles *FinancialClerk* and *Supervisor* at the same time. However, if he has executed *checkInternalRating* in the *FinancialClerk* role, he can never in the future execute *verifyRating* in the *Supervisor* role for the same customer *cst*.

ObjectBasedSoD

SystemConstraints

$$\forall u : USER; cst : Customer \bullet$$
$$\Box((execOpObj(u, financialClerk, checkInternalRating, cst))$$
$$\Rightarrow (\neg (\Diamond(execOpObj(u, supervisor, verifyRating, cst)))))$$

The specification *OperationalSoD* uses *execSeqOp* (Figure 5.5, E11) to prohibit an employee from executing the critical sequence ($Operation_1$,$Operation_2$,$Operation_3$). The specification *OperationalObjectBasedSoD* uses *execSeqOpObj* (Figure 5.5, E12) to disallow executing the operations of a subprocess on the same object *obj*.

Operational_SoD_Patterns

RBAC_model_schema

$SeqOp : \text{seq } OPERATION$

$$\forall u : USER \mid SeqOp = \langle Operation_1, Operation_2, Operation_3 \rangle$$
$$\bullet \Box(\neg (execSeqOp(u, SeqOp)))$$

$$\forall u : USER; obj : Object \mid SeqOp = \langle Operation_1, Operation_2, Operation_3 \rangle$$
$$\bullet \Box(\neg (execSeqOpObj(u, SeqOp, obj)))$$

The specification *OperationalSoD*, on the LAP case study, prohibits an employee from executing the critical sub-process (*enterApplicationData*, *verifyRating*, *verifyTransfer*). The

specification *OperationalObjectBasedSoD* disallows executing the operations of the rating sub-process (*checkInternalRating*, *checkExternalRating*, *verifyRating*) on the same customer application *ctr*.

OperationalSoD

$$
\begin{array}{l}
SystemConstraints \\
SeqOp : \text{seq } OPERATION \\
\hline
\forall\, u : USER \mid \\
\quad SeqOp = \langle enterApplicationData, verifyRating, verifyTransfer \rangle \\
\quad\quad \bullet\ \Box(\neg\ (execSeqOp(u, SeqOp)))
\end{array}
$$

OperationalObjectBasedSoD

$$
\begin{array}{l}
SystemConstraints \\
SqOO : \text{seq } OPERATION \\
\hline
\forall\, u : USER; ctr : Customer \mid \\
\quad SqOO = \langle checkInternalRating, checkExternalRating, verifyRating \rangle \\
\quad\quad \bullet\ \Box(\neg\ (execSeqOpObj(u, SqOO, ctr)))
\end{array}
$$

5.5.5 Specification of Delegation Policies

The specification of the delegation policy is composed of two parts. In the first one, we extend the RBAC model with the delegation concepts. We specify the different types and characteristics of delegation (cf. Section 5.2.3) using a Z schema and we include this schema in the defined RBAC policy by means of schema inclusion. In the second part, we define the delegation operations that correspond to the actions executed by the delegator at runtime.

Specification of the delegation system

We specified user-to-user (U2U) and user-to-role (U2R) delegations combined with permission delegation (PD) and role delegation (RD). We obtained four types of delegation (i.e., U2U4PD, U2U4RD, U2R4PD, and U2R4RD) which are specified in the Z schema *RBACDelegation* which is refined from the *RBAC* schema. We also specified monotonicity and delegation-level characteristics.

In the declaration part, we specify the relations that connect the delegator, the delegated permission/role and delegating user/role. In the constraints part, we specify the constraints that should be satisfied during the delegation operation. As an example, for U2R4PD (Figure 5.9), the delegator should not be a member of delegating role and the delegated operation should not be already assigned in the delegating role.

RBACDelegation

$RBAC$

$U2R4PD : USER \times ROLE \times OPERATION \leftrightarrow ROLE$

$TransferPD : USER \leftrightarrow OPERATION$

$MultiStepPD : USER \leftrightarrow OPERATION$

$GrantPD : USER \leftrightarrow OPERATION$

$SingleStepPD : USER \leftrightarrow OPERATION$

...

$\forall u1 : users; r1 : roles; op1 : operations; r2 : roles$

$\quad | (r2, op1) \notin RoleOperation \land (r2, u1) \notin RoleUser$

$\quad\quad \bullet ((u1, r1, op1), r2) \in U2R4PD$

...

Figure 5.9: Specification of the delegation

Specification of the Delegation Operations

We specify the delegation operations that correspond to the actions that the delegator executes at runtime. The designer should specify each delegation operation by means of a *Z operation schema* and define the input parameters and pre- and post-conditions. The pre-conditions are constituted by the constraints defined in the operation schema and the constraints related to the operations which are defined in the delegation schema. Delegation operations are executed only if their preconditions are satisfied. The post-conditions are the conditions that should be satisfied after executing the delegation operation. The designer should first update the delegation system by defining the new state of the delegation relation. Second, the designer should choose the type of delegation by updating the relations dealing with the delegation characteristics: monotonicity and delegation level.

For illustration, Figure 5.10 presents the operation schema $U2R\_PD$, which specifies the operation user to role for permission delegation. The defined preconditions verify if the delegator is allowed to delegate the corresponding permissions. First, we verify that the role containing the delegator and the delegating role are not defined as exclusive roles in static separation of duties property (cf. Section 5.5.4) [PreC1]. Second, we verify that the user (delegator) is the owner of the delegated permissions [PreC2], or he already received, in a multi-step manner, the corresponding permissions or any whole role containing that permissions [PreC3, PreC4]. We verify also that the delegator does not delegate, in a transfer manner, the corresponding permissions and any whole role containing that permissions [PreC5, PreC6]. For the post-conditions [PostC1, PostC2], the designer updates the system state and specifies that the current operation is transferred in a grant manner.

U2R_PD

$\Delta RBACDelegation$
$u1? : USER; op1? : OPERATION$
$r1? : ROLE$
$r2? : ROLE$

$(r1?, r2?) \in ExclusiveRole$	[PreC1]
$(r1?, op1?) \in RoleOperation$	[PreC2]
$\lor ((u1?, op1?) \in MultiStepPD$	[PreC3]
$\lor (\forall r : roles \mid (r, op1?) \in RoleOperation$	
$\quad \bullet (u1?, r) \in MultiStepRD)$	[PreC4]
$((u1?, op1?) \notin TransferPD$	[PreC5]
$\lor (u1?, r1?) \notin TransferRD)$	[PreC6]
$U2R4PD' = U2R4PD \cup \{((u1?, r1?, op1?), r2?)\}$	[PostC1]
$GrantPD' = GrantPD' \cup \{(u1?, op1?)\}$	[PostC2]

Figure 5.10: Specification of the delegation operation

5.5.6 Ensuring Consistency and Conflict Resolution

To ensure consistency, we verify that the specification of the base RBAC system and its constraints does not contain any contradiction. We do so by using Z-EVES to prove an initialization theorem [192], which states that there exists a state *SystemInit* of the RBAC system and the RBAC extension for delegation that fulfills all constraints.

Theorem *ConsistencyTheorem*
$\exists\, SystemConstraints \bullet SystemInit$

Using Theorem proving, we verify also the preservation of the consistency of the RBAC delegation system after a delegation operation, using a pre-condition theorem [128]. One may still argue that SoD constraints can introduce other inconsistencies. However, as we use a prohibition-based approach [5] (defined by the axioms $\neg\ \varphi$ and $\varphi \Rightarrow \neg\ \psi\ \ldots$), SoD properties are safety properties and therefore cannot produce any inconsistencies into an already consistent RBAC system. If a SoD constraint is evaluated to *True*, the user is prohibited from executing the respective operation. Another reason for choosing the prohibition-based approach is that we cannot force the users to do certain actions to comply with the constraints; we can only prohibit them from executing operations that may break the constraints.

The specification of the separation-of-duty properties on top of RBAC can conflict with a model allowing for the delegation of authority through role and permission transfer [171].

As a simple example, assume that the users $u1$, $u2$ already hold the roles *manager* and *supervisor* respectively. We specify static separation of duty between these two roles, i.e., the two roles must never be assigned to the same user. The delegation of the role or part of the role of the user $u1$ to the user $u2$ would result in a conflict with the separation-of-duty property.

Our solution is based on stating that separation of duties properties always have precedence over other properties [171]. There are two types of conflicts that should be detected and resolved. The first conflict is related to the static separation of duties (as the example presented above). We propose to verify before each delegation operation that the role of the delegator and the delegated role are not defined as mutually exclusive roles. We can verify this through a pre-condition defined in the delegation operation Z schema (see [PreC1] in Figure 5.10 for the LAP example).

For dynamic separation of duties, we present as an example the simple dynamic SoD (the solution concept is the same for the other dynamic SoD). Suppose that we define a simple dynamic SoD constraint between the role $r1$ and $r2$. At runtime, a user $u1$ delegates his role $r1$ to the user $u2$ that belongs to the role $r2$. A conflict is detected, the user $u2$ can execute the operation of the role $r1$ and $r2$ at the same time. We propose a solution on the implementation level: we can verify in all SoD properties that the user does not receive the permission to execute the corresponding action by a delegation operation.

5.6 Aspect-based Enforcement of Access Control Policies

In this section, we present the mapping of the formally specified security policies into enforcement aspects. First, we present the mapping of Temporal$\mathcal{Z}$ predicates into domain-specific Alpha predicates used in the pointcuts of enforcement aspects. Then, we outline the generation schema of aspects that enforce SoD policies and also of aspects that enforce delegation policies. We developed a prototype implementation that incorporates the presented mapping and generates enforcement aspects from the formal specification and deploys them into the Alpha runtime. The prototype is a proof of concept: Efficiency issues are not addressed in this version.

5.6.1 Aspect Language: Alpha

The enforcement of access-control policies is an example of crosscutting concerns. In a conventional implementation, individual application's execution actions that are relevant for the enforcement of a security property, e.g., calls to `assignUserToRole` in LAP, may be brought to being by the execution of code scattered all over the application's module structure. If we want, e.g., to enforce the ObjDSoD property stated in *ObjectBasedSoD* (Section 5.5.4), a check needs to be performed before any call to `verifyRating(cst)` to

make sure that (a) the user performing the operation is in the *Supervisor* role (basic RBAC) and (b) there has not been a call to `checkInternalRating(cst)` by the same user in the role *FinancialClerk*. To enable such reasoning about the past of the execution, some bookkeeping needs to be further performed after each `checkInternalRating` call.

The aspect-oriented programming language Alpha [156] uses logic queries (a subset of valid Prolog queries [187]) as pointcuts to reason about related actions in the program execution. Alpha has a rich joinpoint model: its pointcut queries are run against several models of the program execution including the static program structure (AST), the complete history of the execution (*execution trace*), and the object store at each moment in execution.

The availability of the execution trace is one of the features that distinguishes Alpha from other AO languages. The execution trace can be thought of as a list containing *representations* of all joinpoints that occurred in the execution of the program so far. In Alpha, this list structure is only implicit. Instead, all joinpoint representations carry a time stamp which corresponds to their position in that listing. There are different types of joinpoints corresponding to different types of execution events of an object-oriented program, such as method calls, or field accesses. For each type of joinpoint, there is a relation that has a time stamp as its first argument, the joinpoint specific information as further arguments. Method calls, e.g., are stored in the database as pairs of `calls/5` (i.e., `|/n|` specifies the arity of the relation.) and `endCall/3` facts denoting the beginning respectively the end of a method call. For example, `calls(ID,ExpID, Receiver, Method, Arg)` describes a method call execution action that occurred at time stamp `ID`, the method `Method` is called on the receiver `Receiver` with `Arg` as the input argument; `ExpID` identifies the lexical position of the expression that caused the action. The unary relation `now/1` has – at every point in the execution – only one fact in the database that contains the current time stamp as its argument. The relation `before/2: T` $\rightarrow$ `T` holds if the first time stamp corresponds to an execution action that occurred before the one corresponding to the second time stamp.

Another important model is the store model, which can be used to query arbitrary relations between objects: the `store/3` relation ranges over all possible combinations of objects and fields in these objects by defining a relation between object, field name and their value. Using these relations, pointcut queries can be built-up by the usual Prolog connectives: and (`,`), or (`or`, `;`), and negation as failure (`\+`). As in most Prolog dialects, identifiers starting with uppercase letters are considered variables, whereas all names starting with lowercase letters or enclosed by single quotes (') are considered constants.

The following properties of Alpha are especially useful in the context of encoding stateful access control policies such SoD policies into aspects: *First*, the availability of the execution trace gives access to all relevant execution actions and associated data in a natural way. Furthermore, the ability to add or remove facts during runtime allows a flexible binding between the data objects and the elements in our domain. For example,

information about the relation between users and roles can be added and removed to the runtime database when this information changes. *Second*, the pointcut language allows to *precisely* and *declaratively* specify temporal relations between actions and relations between objects that are part of or related to these actions. This avoids the need for manually implementing the bookkeeping logic for relating this data. *Third,* the possibility to define new predicates/relations in ALPHA allows building up an abstraction of predicates specific for our domain – access control policies.

5.6.2 Mapping Temporal$\mathcal{Z}$ Predicates to Domain Specific Alpha Predicates

A library of domain specific predicates has been defined in ALPHA to facilitate the process of mapping formal specifications to enforcement aspects. These predicates are used in the pointcuts of these aspects. The library includes a Prolog predicate for each temporal operator; RBAC and SoD specific predicates are also part of the library.

The temporal operator $\Leftarrow$ is mapped to the Prolog predicate `before/2`. The operator $\Diamond$ is mapped to the Prolog predicate `eventually/2` which verifies that the first time stamp corresponds to an action after the action corresponding to the second time stamp. Predicates `justBefore/2` and `next/2` correspond to $\ominus$ and $\bigcirc$, respectively. The latter are restricted versions of `before/2`, respectively `eventually/2`: not only does the first time stamp correspond to an action before, respectively after, the second time stamp; there is also no action in-between. The operators $\mathcal{U}$ and $\mathcal{S}$ are mapped to `until/2` and `since/2` predicates, respectively. There is no need for a predicate corresponding to $\Box$; the use of time stamp variables in Prolog predicates states that a property should hold at any time, i.e., always.

There is also a mapping of the logical conjunctives, e.g., the formal conjunction operation ($\wedge$) is mapped to the *and* operator (`,`) of Prolog. Any parameter declared in the formal specification with the quantifier for all ($\forall$) is translated to a Prolog variable.

An example of an RBAC/SoD specific ALPHA predicate is the `callingUser/2` predicate. It makes the user who called an operation accessible. Its implementation is application specific; it depends on the way that user information is stored in the underlying business code.

Another predicate is `verifyRoleAccess/3` which checks that a user belongs to a role that contains an operation using the predicates `roleUser/2` and `roleOperation/2`. These predicates define the relation between roles and users, respectively roles and operations.

5.6.3 Aspects for SoD Properties

For each formally specified SoD property, an ALPHA aspect is generated. This aspect uses an `around` advice to skip the execution if the property does not hold.

Simple Static SoD

The generation of aspects for SSoD properties maps each pair of predicates *ExclusiveRole(r1,r2)* and *include(user,r1)* to the following conjunction of ALPHA predicates: `now(T), calls(T,_,_,assignUserToRole,<User,r1>), roleUser(r2,User)`. For illustration, the aspect generated from the specification *StaticSoD* in Section 5.5.4 is shown in Listing 5.1. The pointcut matches in two cases (note the use of the Prolog operator or). First (lines 3–4), `assignUserToRole` is called to assign an arbitrary user (the parameter `User`) the role `manager`, while that user has the role `supervisor` (see the query `roleUser` in line 4). Second (lines 5–6), `assignUserToRole` is called to assign an arbitrary user the role `supervisor`, while that user is a member of the role `manager` (see the query `roleUser` in line 6). If none of these execution states is reached, the SSoD property is violated; hence, the around advice displays an error and skips the execution of the intercepted method (no call to `proceed`).

```
1 class SimpleStaticSoD {
2   void around
3      now(T), calls(T,_,_,assignUserToRole,<User,manager>),
4      roleUser(supervisor,User);
5      now(T), calls(T,_,_,assignUserToRole,<User,supervisor>),
6      roleUser(manager,User).
7   {
8     System.out.print("Violation of a SSoD property");
9    }
10 }
```

Listing 5.1: Aspect for SSoD

Simple Dynamic SoD

The generation of aspects for SDSoD properties maps the formal predicate $active(u, r)$ to the following conjunction of ALPHA predicates:
`calls(T,_,_,Op,_),callingUser(T,u),verifyRoleAccess(u,r,Op)`. To verify a SDSoD property for two exclusive roles `r1`, `r2` for any user, the pointcut of the generated aspect should be triggered whenever there is a pair of time stamps (`T1`,`T2`), such that operations `Op1` and `Op2` are called by `User` at `T1`, respectively `T2`, whereby `Op1` and `Op2` are associated with roles `r1` and `r2` respectively. If such a pair is found, and `T1` or `T2` denotes the current time stamp `(now(T1);now(T2))`, a violation of the SDSoD constraint is found. If none of them denotes the current time, the violation must have been found earlier. For illustration, Listing 5.2 shows the aspect generated from the specification *DynamicSoD* in Section 5.5.4, which states that any user that is a member of the two exclusive roles *Teller* and *FinancialClerk* cannot activate them at the same time.

```
1 class SimpleDynamicSoD {
2   void around
3
4 %One of timestamps is the current time
5     (now(T1);now(T2)),
6
7 %Translate from the predicate active(User, teller)
8      calls(T1,_,_,Op1,_), callingUser(T1,User),
9      verifyRoleAccess(User,teller,Op1),
10
11 %Translate from active(User,financialclerk)
12     calls(T2,_,_,Op2,_), callingUser(T2,User),
13     verifyRoleAccess(User,financialClerk,Op2).
14    {
15      System.out.print("Violation of a SDSoD property");
16    }
17 }
```

Listing 5.2: Aspect for SDSoD

Object-Based SoD

The specifications of ObjSoD properties use the formal predicate $execOpObj(u, r, op, obj)$. In Section 5.5.4, this predicate is used twice in *ObjectBasedSoD* (with concrete bindings of *op* to *checkInternalRating*, respectively *verifyRating* and of r to *financialClerk*, respectively *supervisor*). The evaluation of $execOpObj(u, r, op, obj)$ to true (`E10` in Section 5.5.2) requires that $execOp(u, r, op)$ is evaluated to true for any *obj* in the set of objects on which *op* operates ($obj \in OperationObject(op)$); The predicate $execOp(u, r, op)$, in turn, is true if $active(u, r)$ is true and *op* is one of the operations assigned to r (`E9` in Section 5.5.2). Following this reduction of the evaluation of *execOpObj* to *active*, the two uses of *execOpObj* in *ObjectBasedSoD* are mapped to the conjunction of Alpha predicates shown e.g., in lines 4–6 and 8–10 of Listing 5.3. The condition $obj \in OperationObject(op)$ used in *execOpObj* is mapped to having the `calls` predicates in the pointcut explicitly relate *op* and *obj*.

This listing shows the aspect generated for the *ObjectBasedSoD* property specified in Section 5.5.4: a user who has executed *checkInternalRating* as a *FinancialClerk* can never in the future execute *verifyRating* for the same customer as a *Supervisor*. The pointcut is triggered whenever a user in a supervisor role is about to verify the rating (at the present time `T2`) and the same user has checked the internal rating of the same `Customer` as a financial clerk at `T1` in the past (`eventually(T2,T1)`); in this case, the advice signals a violation and truncates the execution (no *proceed*).

Unlike the conjunction generated for *active* in Listing 5.2, which uses variables `Op1` and `Op2` to express that the corresponding SDSoD property holds for **any** pair `Op1`, `Op2`, concrete operation names are used in the conjunction sequences in Listing 5.3 to reflect the fact that *execOpObj* in *ObjectBasedSoD* refers to concrete operations. Also, in addition to

the `User` variable, the `Customer` variable is used in the `calls` predicate in Listing 5.3 to express that the property should hold for **any** customer object.

```
1  class ObjDynamicSoD {
2    void around
3
4      calls(T1,_,_,checkInternalRating,Customer),
5      callingUser(T1,User),
6      verifyRoleAccess(User,financialClerk,checkInternalRating),
7
8      now(T2), calls(T2,_,_,verifyRating,Customer),
9      callingUser(T2,User),
10     verifyRoleAccess(User,supervisor,verifyRating),
11
12   %checkInternalRating is called before verifyRating
13     eventually(T2,T1).
14   {
15     System.out.print("Violation of a ObjDSoD property");
16   }
17 }
```

Listing 5.3: Aspect for ObjDSoD

Operational SoD

The generated aspect has a pointcut that verifies if the user is about to execute the last operation of a critical sequence of operations. For instance, given the sequence (`enterApplicationData`, `verifyRating`, `verifyTransfer`) passed to the formal predicate *execSeqOp* in *Operational-SoD* in Section 5.5.4, the pointcut shown in Listing 5.4 is generated. It is triggered on a call to the last operation of the sequence and checks if the user has called the other operations of that sequence in the respective order. In this case, a violation is signaled and the last operation is not executed.

```
1  class OperatSoD {
2   void around
3
4     calls(T1,_,_,enterApplicationData,_),
5     callingUser(T1,User),
6     verifyAccess(User,enterApplicationData),
7
8     calls(T2,_,_,verifyRating,_),
9     callingUser(T2,User),
10    verifyAccess(User,verifyRating), eventually(T2,T1),
11
12    now(T3), calls(T3,_,_,verifyTransfer,_),
13    callingUser(T3,User),
14    verifyAccess(User,verifyTransfer), eventually(T3,T2)
15    {
16      System.out.print("Violation of a OpDSoD property");
17    }
18 }
```

Listing 5.4: Aspect for OpDSoD

Operational Object-Based SoD

The generation of aspects for *OpObjDSoD* properties is similar to that for operational SoDs except that the last parameter of the generated `calls` predicates is a variable rather than unbound. This is to reflect that object-based operational SoD properties are specified for **any** $obj \in OBJECT$. By using the same variable in all generated `calls` predicates, Prolog unification ensures that the pointcut is triggered only when the critical sequence is executed on the same object.

5.6.4 Aspects for Delegation Policy

There is one aspect for each delegation operation which enforces the corresponding pre-conditions and delegation constraints. The pseudo code in Listing 5.5 shows the overall structure of such an aspect.

The pointcut (line 2) is composed of the ALPHA predicate calls that is used to intercept calls to the method in the functional code which corresponds to the formally specified delegation operation. The advice (lines 2–20) associated with the pointcut has the type *around*, i. e., it is executed instead of the intercepted method calls.

```
1  class DelegationOperation {
2     void around calls (N, _, _, delegateOperation,obj){
3
4        //Evaluate pre-condition in operation schema
5        //Evaluate conditions in delegation schema
6         if (zOperatorMethod(parameters, systemState)){...}
7         if (constraint_i()) { ... }
8
9        //Evaluate all defined conditions
10       if (allConstraints) {
11            //Execute of the delegation Operation
12            proceed();
13            //Update post-conditions
14           updateSystemState();
15       }
16       else
17       {
18           //An exception is handled
19       }
20    }
21    public boolean constraint_i() { ... }
22    public void updateSystemState() { ... }
23 }
```

Listing 5.5: Template of the generated Alpha aspect

In the advice body, each Z constraint is translated to a conditional statement. In order to easily translate the Z constraints to Java conditions, we implemented a Java library for Z. It contains the implementation of the most used Z operators and mathematical objects

as Java methods. Two types of conditions are generated depending on the Z constraints. Constraints of the first type are simple conditions, generated from formal constraints without quantification. These constraints are checked by calling the corresponding method in the Java Z library zOperatorMethod (line 6), which takes the system state as parameter. Helper methods, denotated by the constraint_i, are generated for each formal constraint that uses a quantification operator (lines 8 and 21).

If all generated conditions are fulfilled (line 10), the intercepted operation will be executed using the keyword proceed (line 12), which executes the intercepted method call. Otherwise, the aspect prohibits the execution of the operation (lines 16–19). In addition, the generated aspect comprises a method (line 22) to update the system state according to the specification of the post-conditions. This method is called after the proceed when the generated preconditions are fulfilled (line 14).

5.7 Related Work

We discuss next related work on the specification and the enforcement of access control policies and especially works on RBAC, separation and duties and delegation policies.

5.7.1 Specification of Access Control Policies

Several works have addressed the specification of role-based access control using formal methods and essentially using logic such as temporal logic [146, 172] or Z notation [1, 197]. Many other formal methods have been also used to specify access control likes Petri Nets [174], graphs [119], process algebra [37], and automata [71, 130, 173].

To cover all aspects of access control policies, we use a new formal language called *Temporal$\mathcal{Z}$*, which is an extension of the Z notation with temporal operators. The advantages of *Temporal$\mathcal{Z}$* for specifying these policies are detailed in Section 5.5.2. Several works proposed to extend Z with formal methods supporting temporal features, such as linear temporal logic [62], temporal logic of action [127], interval temporal logic [40], etc.

The most related work to ours is that of Duke and Smith [62], which extends Z to include temporal logic operators. Unlike our *Temporal$\mathcal{Z}$* language, Duke and Smith [62] just use temporal operators to reason about (possibly infinite) sequences of Z schema states, where two consecutive states correspond to "before and after" states of an already specified Z schema operation. In our work, we extend the Z mathematical notation with temporal operators. Thus, temporal operators can be applied to any Z formula, which is not necessary about Z states. Moreover, the semantics that we provide for temporal formulas within the framework of Z enables us to use the existing Z tools for syntax and consistency checking as well as theorem proving of *Temporal$\mathcal{Z}$* specifications.

Fidge [70] propose to specify timing requirements expressed in real time logic (RTL) in Z specifications. He extends the Z notation with most of RTL operators. The proposed approach is based on the execution history as defined in Duke's work [62]. An event is defined as an operation, or a state attribute becoming true or false. An occurrence function is defined which returns for each event (i.e., input argument) the sequence of all time which it occurred or the i^{th} occurrence of the event (i.e. with i as second input argument). In addition, a verification approach of real-time properties based on this extension is defined. The proposed approach is related to our temporal model which defines for each time the set of security predicates that hold at this time. Unlike our approach, no semantic definition of the new language is proposed. In our temporal model and the evaluation functions, we defined formally the semantics of our Z extension. In addition, we proposed a verification approach of the temporal properties using Z theorem prover.

In the following, we concentrate on approaches, which are related to ours with respect to the specifications of access control policies. These works use different formal languages and methods.

Based on automata theory, Schneider [173] defined security policies that are enforceable by execution monitoring. The author proposed security automata which are formed of Büchi automata that recognize safety properties. Bauer et al. [130] extended this idea by defining edit automata which are formal machines that modify the program action via the operation of sequence truncation, insertion and suppression of actions. Fong [71] defined the Shallow History automata which track only the shallow history previously granted access events. With this automata, it is still possible to express policies like the Chinese Wall policy [178]. These previous works are more theoretical and have not been applied for access control policies. Also, they do not provide any rules and algorithms for translating specifications to code. The work of Basin et al. [24] is more practical and proposes to specify security automata using CSP-OZ.

Using temporal logic, Mossakowski et al. [146] specify RBAC and essentially dynamic separation of duties. This work does not cover all RBAC constraints and it does not support formal verification as no tool has been used for verification. Similar to this work, Schaad et al. [172] propose a model-checking approach for analysis of delegation and revocation in RBAC system as well as the formal specification of the different types of separation of duties. In this approach, no verification approach is defined to verify the absence of the contradiction between formal constrains. In contrast to these works, we use Z notation to formally specify RBAC and delegation properties. In addition, we verify, with the Z/EVES theorem prover, the consistency and other application specific properties.

Z notation [181] and the theorem prover tools provide an effective means for a precise and unambiguous specification of secure software systems. In [1], the authors show formal Z specifications of several flat RBAC models. They define formally the RBAC concepts (i.e.

roles, permissions, principals,...), the relationships between them, and some constraints but they do not address the separation of duty properties. In [197], the authors specified the RBAC model using Z and verified the consistency of the specified model using theorem proving. This approach is similar to ours w.r.t. specification but it supports only the simple static and dynamic SoD, which can be specified with Z notation as presented in our case study. The other types of separation of duties, which require temporal constraints, are not supported. We note also that the previous works do not address the translation of formal specifications to code.

Using an extension of the π calculus, Braghin et al. [37] define an RBAC scenario for concurrent systems. This approach does not support the specification of SoD properties. Compared to the π calculus, Temporal$\mathcal{Z}$ is more declarative and can be *translated* to code easier, because Z specifications are near to the implementation unlike other more abstract formal languages.

Ahn and Sandhu developed the RSL 99 [6] language and its extension RSL 2000 [7] language to specify the SoD properties. In these works, the authors describe the formal syntax and semantics of this language by defining an automatic translation to a first-order predicate logic called RFOPL. Moreover, the authors proposed a new form of static and dynamic SoD by defining the set of roles, permissions, and users. In their work, the authors are interested only in the simple separation of duties properties and they do not care about the other constraints in role-based systems. Moreover, the defined language cannot be used to specify the different types of SoD properties as it does not support the concept of object, sequence and time of execution.

Unlike our proposal, these works do not present any mechanism to enforce the access control properties in application code.

5.7.2 Enforcement of Access Control Policies

Several works focused on the development of the translation of specified access control to code. Some works use UML and OCL for that purpose [23, 99, 190]. Others proposed a new notation to define easily the access control requirements [7, 150].

The UML-based approaches use different extension mechanisms and various UML diagrams to model access-control constraints. The most related work to ours in that thrust is the one of Basin et al. [23]. The authors propose a model-driven security approach to specify access control authorization and they generate automatically the application model including access control support. In that work, a new security modeling language is defined. It combines system models and their security requirements using dialects. In addition, several transformation functions are defined. The latter produce model elements that support

access control for distributed component-based applications and for web applications. However, only RBAC system constraints are addressed; temporal constraints (including separation of duties) cannot be specified because of OCL limitations. Also, there is no support to verify the consistency of the system and prove that the specification does not contain any contradiction. At the implementation level, we generate modular and executable code to enforce the RBAC constraints whilst the work in [23] generates an incomplete code for some component model, which is tangled with the rest of the application code.

In [150] Neumann presented the design and implementation of an RBAC-service called xoRBAC. This service provides a policy monitor for RBAC policies which can be applied to applications providing C or Tcl linkage. xoRBAC is implemented with XOTcl. It supports high-level modeling of roles, permissions and constraints, and provides RDF import/export for access-control data. This approach supports context constraints, which makes it flexible and dynamically extensible. This approach does not define a dedicated policy language for specifying RBAC and separation of duties policies. No verification mechanism or tool can be used to verify that no contradiction exists between constraints. The enforcement mechanism works only for web-based mobile code that is implemented with an object-oriented scripting language.

5.7.3 Access Control using Aspect-Oriented Techniques

Many works [51, 158, 180, 188, 194] profit from the advantages of aspect-oriented techniques for modeling and enforcing access control policies in a modular way. In the following, we present the most related works to ours.

In [188], Verhanneman et al. proposed an approach for enforcing context-based access control policies. They implemented a modular access control service for supporting application-specific policies. This service intercepts the critical operations using aspects, calls the authorization engine, and enforces the respective constraints. The approach is based on two concepts: access interfaces and view connectors. The first is an abstraction layer that reflects the information that is relevant for access control. The view connector binds the access interface to the application and uses the access control decision defined by the authorization engine each time an application event occurs. A prototype is implemented in JAC [161] and CaesarJ [14]. This approach allows only to enforce simple access-control constraints defined in Ponder [56] or in XACML [81]. The separation of duties properties and administrative operations cannot be specified by these security mechanisms. In addition, compared to Alpha, the aspect languages used for implementing the service are based on less powerful pointcut languages. This leads to less modular aspects especially when temporal constraints are to be enforced [156].

Song et al. [180] proposed an aspect-oriented modeling approach to design role-based access control. This approach is based on UML and OCL. It allows to design access control

features by the verifiable composition of the aspects and primary model. This composition yields a design model in which access control features are integrated with the other application features. The proposed approach covers only the modeling phase and it can specify only the basic system constraints using OCL. No temporal constraints are supported for specifying dynamic separation of duties properties. In addition, the security constraints defined with OCL cannot be rigourously verified with respect to consistency. Moreover, no mechanisms have been proposed to translate the defined aspect model to aspect code.

Mariscal et al. [158] propose a formal framework for enforcing role-based access control model. The authors formalize the concept of a role slice (i.e., new visual notation to present roles and permissions in RBAC) by defining a specialized UML class diagram. This diagram contains a list of permissions defined in form of allowed or prohibited methods. Then, they define a formal framework for secure software applications by translating automatically the role slice access control policy into aspect enforcement code. The authors formalized the compilation process in order to integrate AOP generated security code with the application code. However, this work is merely theoretical and does not provide any support to generate aspects. Moreover, separation of duties properties and delegation policies are not addressed.

5.7.4 Specification and Enforcement of Delegation Policies

Several delegation models have been proposed in the literature [20, 179, 198, 199]. In the following, we present the most related ones to our work.

RBDM0 [20] is a role-based delegation model. It extends the basic RBAC by new components and relations for differentiating the delegated and original member as well as their assignment to roles. RBDM1 [21] is an extension of RBDM0 to support delegation in a role hierarchy. Compared to our approach, RBDM0 does not support permission delegation and transfer delegation. Further, enforcement is not addressed. A prototype implementation of RBDM has been proposed [19]. The implemented code is not well modularized compared to ours using aspect-oriented programming.

RDM2000 [198] is an extension of RBDM0, which supports role delegation. In that work, the authors define a rule-based policy specification language for enforcing delegation. This language is used with GUIs to handle delegation and revocation requests. In our approach, we use the Alpha language based on Prolog which allows to easily manage the delegation components and relations as queries. In addition, we generate automatically the security enforcement code from the formal specification, which avoids errors.

PBDM [199], is a model for role and permission delegation that supports multi-step and grant delegation. Similar to RBDM, this model extends RBAC with new components to define the delegation policy and its characteristics. PBDM is extended to PBDM1 and PBDM2 to allow security administrators to perform the delegation operation and role-to-role delegation. PBDM and its extensions do not support the transfer delegation and the

management of delegation history.

In [179], Sohr et al. specify formally using temporal logic and predicate logic the delegation model. In this work, the authors follow the RDM2000 model and define new concepts and components to manage the delegation in RBAC model. Compared to our approach, the delegation operations and the preconditions to delegate rights are not defined. In addition, no verification was proposed to check if the formal specification contains any contradiction. Also, no enforcement mechanism has been proposed.

Few works have been done on enforcing delegation policies with AOP. In [50], the author extends his access control framework for Struts-based Web applications to include delegation of access rights. The proposed approach is based on the aspect language AspectWerkz used to dynamically enforce delegation policies. Compared to our approach, the automatic generation of delegation aspects from formal specification allows the user to easily specify the delegation model and its characteristics and bridge the gap between specification and implementation. In addition, our approach supports different types and characteristic of delegation that are not considered in [50], such as user-to-role delegation.

5.8 Conclusion

We applied the SEVEN-pro approach on access control policies. Our approach covers the formal specification and enforcement of separation-of-duties and delegation policies on top of RBAC. To easily specify and enforce these policies, we extended *TemporalZ* by domain specific predicates and we proposed some specification patterns for SoD properties. We also extended the ALPHA language by new predicates for easily enforcing specified policies. We also explained the automatic translation of *TemporalZ* specifications into ALPHA aspects.

The efficiency of ALPHA as an aspect language for enforcing access control policies is one of the limitations of this work. ALPHA is very slow and cannot be applied to real systems. It is also hard to guarantee termination of the system execution given that all ALPHA predicates (facts) are stored in the database and will be never removed. Also, our approach does not address role hierarchy, which is an extension of flat RBAC to structure the roles in a large organization. This extension is a key concept for delegation policies in reals systems.

In the next chapter, we apply the SEVEN-pro approach on the specification and the enforcement of temporal properties in Web service compositions. We combine the XTUS language within timed automata for formally specifying absolute and relative temporal properties. These specifications are automatically translated to aspects in AO4BPEL, which is aspect-oriented extension for BPEL.

Chapter 6
Specifying and Enforcing Temporal Properties

6.1 Introduction

Web service composition is being used in various domains, such as *business-to-business integration* (B2B) or *enterprise application integration* (EAI). Current composition approaches focus on the specification of a set of interactions between Web services together with the flow of control and data around them and often neglect other important concerns such as temporal properties or quality of service.

Temporal properties are essential in several Web service composition scenarios as time plays a crucial role in business processes and B2B interactions. Current Web-service composition approaches and languages, such as the standard WS-BPEL [98], provide language constructs for managing time. E.g., WS-BPEL provides special activities such as *onAlarm* or *wait* that can be used to implement temporal properties. However, the WS-BPEL approach has several limitations: (1) to express temporal properties, one needs to program them manually in the process specification using WS-BPEL activities; (2) only simple time expressions such as timeouts and durations may be expressed; (3) the correctness of a Web service composition after manual insertion of time related activities cannot be verified; (4) the process code implementing the temporal properties is mixed and tangled with the process logic.

To recap, current Web service composition approaches do not allow a declarative and separate specification of temporal properties. This leads to maintenance problems as the designer of the composition cannot modify the temporal properties without understanding the whole process logic and identifying the process activities implementing a given temporal constraint. Further, Changes to temporal properties cannot be done at runtime, i.e., one has to stop the composite Web service, undeploy the corresponding process, change the process, and redeploy it. In addition to the maintainability problem, there is a correctness problem that arises when not using a formal language. For instance, when temporal properties are

implemented using BPEL activities one cannot verify if there are inconsistencies or deadlock states because of lacking formal verification tools.

In this chapter, we address the limitations explained above. We apply the SEVEN-pro approach on temporal properties in Web service compositions. The contributions presented in this chapter are published in [104].

The remainder of this chapter is organized as follows. Section 6.2 introduces a running example: a travel agency scenario. Section 6.3 describes the proposed approach for implementing temporal properties. Section 6.4 presents the formal language and the specification patterns used for specifying temporal properties in Web service composition. Section 6.5 describes the mapping of the formal specifications to aspect code, and Section 6.6 presents an overview of the research prototype of this mapping. Section 6.7 reports on related work and Section 6.8 concludes this chapter.

6.2 Example: Travel Agency Scenario

We introduce an example of Web service composition in the context of a travel agency scenario. The example is inspired from [84].

We consider two business processes that contain activities representing either business tasks or interactions between Web services and customers: the travel package search process and the travel package booking process. Both processes compose services of airlines and hotels to find (resp. book) vacation packages whereby five partners are involved: the travel agency, customers, an airline, a hotel chain, and a payment service.

Once a customer sends a travel request to the travel agency, the travel-package search process interacts with the information systems of the airline companies to find flights that match the customer's needs. In addition, this process searches for available hotel rooms by interacting with the information systems of several hotel chains. After that, the travel agency sends the response as a set of offers combining the respective flight and hotel responses. Next, the customer could select one offer. This step launches the travel-package booking process, which interacts with the airline and hotel services to verify availability. Upon availability, this process asks the customer to confirm the purchase of the travel offer and to provide payment details (e.g., using a credit card). In the last step, this process performs the booking of the different components of the travel package and sends to the customer a confirmation by e-mail including all trip documents.

We present in the following some temporal constraints, that are important in the context of the travel agency example.

- **C1:** The customer should pay the selected offer by providing his credit card data within 30 minutes after the reservation step. Otherwise, the reserved offer will be canceled.

- **C2:** The customer is allowed to do only 3 failed payment trials and these trials should occur during 5 minutes.
- **C3:** The customer can cancel a travel reservation 7 days before his travel at the latest.
- **C4:** The customer can change his travel reservation only 2 times. Changes are only allowed between 1 day and 5 days after the reservation date.
- **C5:** If the booking is done in a special period, a discount is given.

6.3 The Approach in a Nutshell

We applied the three steps of the SEVEN-pro approach for enforcing temporal properties in Web service compositions, as shown in Figure 6.1.

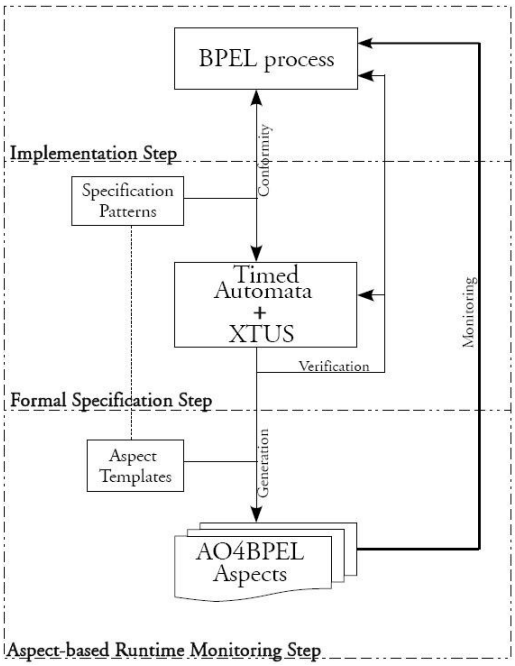

Figure 6.1: Specifying and enforcing temporal properties

In the *formal specification* step, the designer formally specifies the temporal properties using XTUS-Automata, which is a combination of Timed Automata [12] with the eXtended Time Unit System [35]. Thereby, the designer should follow and respect so-called specification patterns, which we provide for different types of temporal properties. In addition,

the activity names and the parameters used in the formal specification step should be the same as those used in the WS-BPEL process. After this step, formal verification of deadlock freedom and other properties can be done using existing model-checking tools such as UPPAAL [58].

In the *implementation* step, the designer defines the Web service composition using WS-BPEL but without implementing any temporal properties.

In the *enforcement* step the designer uses our aspect-generator tool to automatically generate enforcement code consisting of AO4BPEL aspects. These aspects enforce the temporal properties at runtime and ensure that they are satisfied. If that is the case, the process activities are executed; otherwise, the aspect prohibits the execution of the monitored activities, and throws an exception.

6.4 Formal Specification

In this section, we first present the timed automata and the extended time unit system. Then, we define the syntax and the semantics of our new formal language XTUS-Automata combining the two previous languages. Finally, we introduce the specification patterns that enable expressing temporal properties in Web service composition using XTUS-Automata.

6.4.1 Formal Languages

For the sake of a better understanding of XTUS-Automata, we review the concepts of timed automata and extended time unit system.

Timed Automata

Timed Automata (TA) [12] specify the behavior of real-time systems over time. They extend finite-state automata with time constraints using continuous clock variables. Clocks can be reset at certain transitions and their values can be used in constraints for enabling/disabling transitions.

Definition1: Timed Automata
A Timed Automaton $\mathcal{A}$ is a 6-tuple (L, l_0, E, C, Inv, T):

- L is a finite set of locations,
- $l_0 \in$ L is an initial location,
- E is a set of events/actions,
- C is a finite set of clocks,

- *Inv* is a mapping that labels each location $l \in L$ with some inequalities of the form c $\leq$ n, for some clock c and integer n,
- $T : L \times E \times 2^C \times \Phi(C) \times L$ is the transition relation.

$(l,\ e,\ \lambda,\ \varphi,\ l')$ is a transition from location l to location l' on event e. λ gives the clocks to be reset with this transition, and φ is a clock constraint over C that specifies when the transition is enabled. The set of clock constraints $\Phi(C)$ is defined by the following grammar:
$\varphi := c_1 \sim n \mid c_1 - c_2 \sim n \mid \neg\varphi \mid \varphi \wedge \varphi$
where $\sim \in \{<, \leq\}$, $c_1,\ c_2 \in C$, and n $\in \mathbb{N}$.

A transition $(l,\ e,\ \lambda,\ \varphi,\ l')$ $\in$ T is enabled if the control is at location l, the clock constraint φ is satisfied and the event e is enabled. After taking the transition, the control moves to location l' and the clocks in λ are reset.

Clock constraints are evaluated over clock valuations. The clock valuation $v : C \to \mathbb{R}_{\geq 0}$ is a function from the set of clocks to the positive reals. A valuation v satisfies the clock constraint φ, denoted by $v \vDash \varphi$, and defined as follows, where c is a clock and r is a positive real.
$v \vDash c < r$ iff $v(c) < r$
$v \vDash c \leq r$ iff $v(c) \leq r$
$v \vDash \neg\varphi$ iff $v \nvDash \varphi$
$v \vDash \varphi_1 \wedge \varphi_2$ iff $v \vDash \varphi_1$ and $v \vDash \varphi_2$

Definition 2: Semantics of Timed Automata
The semantics of a timed automaton $\mathcal{A} = (L,\ l_0,\ E,\ C,\ Inv,\ T)$ is defined as a labeled transition system $\langle S, s_0, \to \rangle$, where

- S $\subseteq L \times \mathbb{R}^C$ is a set of states. Each state s_i is defined as a pair (l_i,v_i) such that l_i is a location and v_i is a clock valuation for C and v_i satisfies $Inv(s_i)$, i.e., $S = \{(l, v) \in L \times \mathbb{R}_{\geq 0} \mid v \vDash Inv(l)\}$
- $s_0 = (l_0,\ v_0)$ is a initial state, where l_0 is an initial location and v_0= 0 for all clocks, and
- $\to \subseteq S \times \{\mathbb{R}_{\geq 0} \cup E\} \times S$ is a transition relation that takes into account the invariants and that is defined as follows:
  - $(l, v) \xrightarrow{d} (l, v + d)$ if
    $\forall\, d' :\in \mathbb{R}_{\geq 0} 0 \leq d' \leq d \Rightarrow (u + d') \vDash Inv(l)$, where $d \in \mathbb{R}_{\geq 0}$,
  - $(l, v) \xrightarrow{a} (l', v')$ if
    $\exists\ t = (l, e, \lambda, \varphi, l') \in T \Rightarrow v \vDash \varphi$ and $v' \vDash Inv(l')$, with $v' = [\lambda \longmapsto 0]v$, where $[\lambda \longmapsto 0]v$ denotes the clock valuation which maps each clock in λ to 0.

Extended Time Unit System

The Time Unit System (TUS) [125] provides a simple way to specify time instants and intervals based on a common way for expressing dates. TUS represents time intervals as sequences of non-negative whole numbers in the form [Year, Month, Day, Hour, Minute, Second**,...**]. For example, `[2009,6,13,22,30,0]` denotes the first second of the minute 30 of the hour 22 on the 13th day of June 2009. Intervals can also be built using the operator *convexity*. For any two time intervals t_1 and t_2, *convexity*(t_1, t_2) is the smallest convex interval that contains both t_1 and t_2. For example, *convexity* (`[2000],[2009]`) denotes the first ten years of the 21st century.

The Extended Time Unit System (XTUS for short) [35] extends TUS and improves its expressive power to define time intervals. In addition to the fixed time instants presented as integers in TUS, XTUS proposes new elements. An *interval element* defines the time interval between two time instants, denoted by [t_i – t_j] which is equivalent to [t_i, $t_i + 1$, $t_i + 2$, ..., t_j]. For example `[5−11]` corresponds to `[5,6,7,8,9,10,11]`. An *anonymous element* corresponds to any possible value of the time unit denoted by "_" (underscore – inspired from the anonymous variable in Prolog). For example, `[2009,_,[1−10]]` denotes the first 10 days in any month of 2009. A *combined element* corresponds to the result of the combination of two different elements (i.e., integer or interval) in any level of hierarchy, denoted by {}. For example, the combined element `[_,{[1−5],[8−12]},_]` denotes every day in every month except June and July (months 6 and 7) of any year.

XTUS supports also some algebraic operations for defining more complex temporal properties. Given r_1, r_2 two terms specifying intervals of XTUS, then $r_1 \cup r_2$, $r_1 \cap r_2$, $r_1 \backslash r_2$ denote respectively the union, the intersection, and the difference of these intervals.

6.4.2 XTUS-Automata Language

To the best of our knowledge, there is no formal language that can be used to specify both relative time and absolute time properties for Web service composition. The combination of XTUS and timed Automata overcomes the limitation of timed automata with respect to absolute time. In fact, XTUS allows to specify temporal properties that use absolute time, while timed automata allow to (1) specify the execution order of the different actions/activities in the process underlying the service composition, (2) specify the maximum and minimum occurred time between the execution of two different actions/activities, and (3) express that certain actions may be repeated n times within a given time period.

In addition, temporal properties specified using XTUS-Automata can easily be translated to code. The translation of XTUS expressions is straightforward, as time is specified using a set of integers and elements (i.e., interval, anonymous, and combined elements). Thanks to the specification patterns and the aspect templates the timed automata can be

translated to code. This will be presented later.

In addition to clock resets and to temporal constraints (see definition 2 in Section 6.4.1), we propose to specify XTUS constraints at the automata transitions (see Section 6.4.1). In this way, an action defined in the business process (i.e. modeled as a timed automated transition) will be executed if and only if two types of constraints defined on the corresponding transition are satisfied: Timed Automata constraints and XTUS constraints. To allow expressing constraints that are relative to the current time, the designer can assume the availability of a special clock *ct* that corresponds to the current time. That special clock cannot be reset.

In order to use XTUS for the formal specification of temporal constraints we have formally defined its syntax and semantics using the Z notation [181].

In the following, we first present the formal definition of the different elements *Element* defined in XTUS (simple time or date value ($\mathbb{N}$), interval ($Interval\langle\langle \mathbb{N} \times \mathbb{N} \rangle\rangle$), anonymous (*Anonymous*), and combined elements ($\mathbb{F}\ UnitInterval$)), which were informally presented in Section 6.4.1. The function *Def* describes the semantics of each element: we associate for each type of *Element* a set of integers that represents the possible values of this element. For example, the function *Def* associates with the element $Interval(n_1, n_2)$ the set of all possible integers between $n1$ and $n2$.

$$
\begin{array}{l}
XTUS = \text{seq}\, Element \\
UnitInterval ::= \mathbb{N} \mid Interval\langle\langle \mathbb{N} \times \mathbb{N} \rangle\rangle \\
Element ::= Anonymous \mid UnitInterval \mid \mathbb{F}\ UnitInterval
\end{array}
$$

$$
\begin{array}{|l}
Def : Element \rightarrow \mathbb{F}\,\mathbb{N} \\
\hline
Def(Anonymous) = \{n : \mathbb{N}\} \\
\forall\, n : \mathbb{N} \bullet Def(n) = \{n\} \\
\forall\, n1, n2 : \mathbb{N} \bullet Def(Interval(n1, n2)) = \\
\qquad \{n : \mathbb{N} \mid n \geq n1 \land n \leq n2\} \\
\forall\, u : \mathbb{F}\ UnitInterval \bullet Def(u) = \{n : \mathbb{N} \mid n \in u\} \\
\qquad \cup \{n, n1, n2 : \mathbb{N} \mid Interval(n1, n2) \in u \\
\qquad \land\, n \geq n1 \land n \leq n2 \bullet n\}
\end{array}
$$

In our formal definition, we consider the time in the form [Year, Month, Day, Hour, Minute, Second] and a time instant is represented by a set of time values (natural numbers) corresponding to the different units. We specify also using appropriate Z schemas the domains of the different time units (i.e., seconds, minutes, etc). The domains allow us to prohibit users from specifying constraints such as `[2009,[1−4],[20−30]]`, as February does not have the days 29 and 30.

An XTUS constraint is satisfied only if the current time *ct* is one of the possible time values corresponding to that constraint (after applying the function *Def*). For that, each

time value in the current time instant $ct(i)$ (year, month,...) defined as natural number ($\mathbb{N}$) should satisfy its respective element $X(i)$ ($Element$) in the XTUS constraints using the function $Verif$ defined below.

The satisfaction function Sf allows to verify that the time values of all units (i.e., seconds, minutes, etc) in the current time instant are satisfied using the function $Verif$.

$$Verif\_ : \mathbb{P}(\mathbb{N} \times Element)$$

$$\forall\, n : \mathbb{N}; e : Element \bullet Verif(n, e) \Leftrightarrow n \in Def(e)$$

$$Sf\_ : \mathbb{P}(\operatorname{seq}\mathbb{N} \times \operatorname{seq} Element)$$

$$\begin{aligned}&\forall\, SeqNat : \operatorname{seq}\mathbb{N}; SeqElem : \operatorname{seq} Element \\ &\quad \mid \#SeqNat = \#SeqElem = 6 \bullet \\ &Sf(SeqNat, SeqElem) \Leftrightarrow (\forall\, i : 1..\#SeqNat \bullet \\ &\quad (Verif(SeqNat(i), SeqElem(i))))\end{aligned}$$

6.4.3 Specification Patterns

We propose specifying temporal properties in Web service compositions using XTUS-Automata based on *specification patterns*. The use of patterns facilitates the specification of complex temporal constraints and allows to structure the specification so that it can be automatically translated to runtime monitoring code.

In our specification model, the Web service composition is considered as a sequence of finite states, which are reached by transitions representing WS-BPEL activities such as the messaging activities *invoke*, *receive*, and *reply* used for invoking a partner Web service and/or interacting with a client. The message transmitted from the caller is denoted by the symbol $!msg$ and the message received is denoted by the symbol $?msg$. For us, only the time or date variables are considered as parameters in the transmitted messages $msg(t_1, t_2, ...)$.

The automata clock can be defined and initialized in three ways. First, the clock can be initialized to zero, as in timed automata. Second, the clock can be an input parameter for a BPEL activity (i.e., defined as message transmitted between Web services). Third, the clock can be defined as an output parameter of some BPEL activity already executed.

In the following, we describe some XTUS-Automata patterns used to define some useful temporal properties that occur frequently in the context of Web service compositions.

Patterns for Duration Properties

These properties allow to specify a fixed duration between the execution of two different activities. An activity must eventually follow another activity within, after, or at the time t.

Figure 6.2 depicts three patterns of this type which specify that the activity e_n should be executed within a fixed time after the execution of the activity e_1. In Figure 6.2a, at the

automata transition of the activity e_1, a clock x is reset and a clock constraint is defined at the transition of the activity e_n. This activity e_n can only be executed if the defined constraint is satisfied. In case the activity e_1 corresponds to a BPEL activity with a date or time as input parameter (see Figure 6.2b), or if the BPEL activity returns a time or date as an output parameter (see Figure 6.2c), these parameters are defined at the corresponding transition instead of the expression for resetting a clock. In these two cases, we use the current time ct to define the clock constraints.

(a) Pattern with clock reset

(b) Pattern with input parameter

(c) Pattern with output parameter

Figure 6.2: Patterns for duration properties

Where $\sim$ denotes $<$, $\leq$, $>$, $\geq$, and $=$.

In Figure 6.3, we apply the pattern shown in Figure 6.2a as duration pattern with clock reset for specifying the constraint C1 defined in Section 6.2. C1 requires that the customer should pay the travel fees the latest 30 minutes after the reservation. We store the time of the reservation step by resetting the clock x at the transition *reservePack*, and we define the clock constraint ($x < 30min$) at the transition *bookTicket*. In case the customer books the ticket of his travel before the end of the 30 minutes period, he will receive a confirmation of his booking; otherwise the reservation will be canceled automatically.

For example, to specify the constraint C3, which states that the customer can cancel his reservation the latest 7 days before his travel, we apply the duration specification pattern with input parameter (Figure 6.3). The transition *reservePack* should be defined as a

transition with input parameters (departure date DD and arrival date AR). In the transition *cancelReservation*, the designer specifies the time constraint ($ct \leqslant DD - 7days$) to express that canceling the reservation cannot be executed if it happens later than 7 days before the departure date.

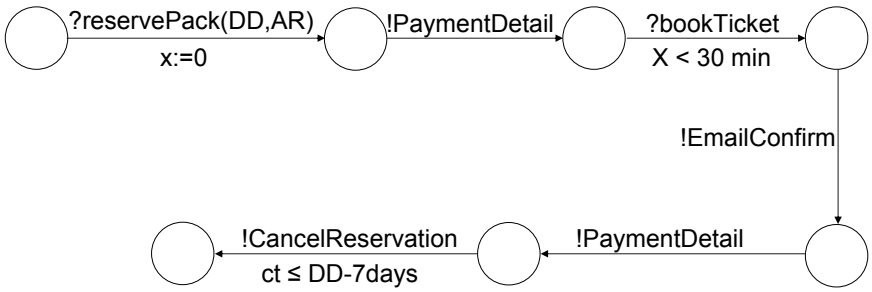

Figure 6.3: Example of a duration property

Pattern for Temporal Properties over Cardinalities

The following properties define that some activity e can be executed successively n times within a fixed time period. Figure 6.4 shows the specification pattern for this type of temporal properties. The designer should specify the repetition of the activity e using multiple transitions (and without using a loop). In the first transition, a reset clock has to be defined to store the time execution of this activity. The other $(n-1)$ activities should be defined as next transitions and can contain some clock constraints depending of the temporal property. In addition, between the transitions corresponding to the activity e, any other transition a_i can be specified.

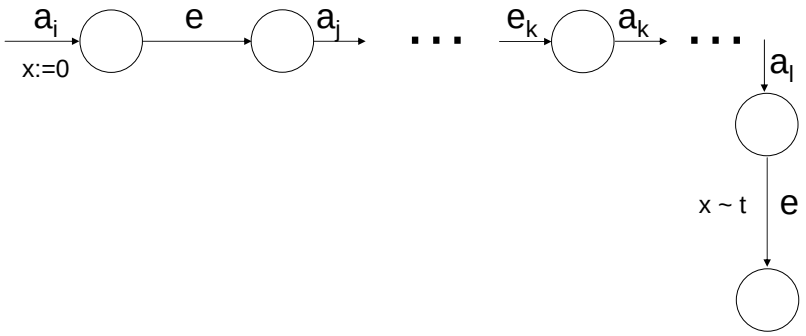

Figure 6.4: Pattern for temporal properties over cardinalities

Figure 6.5 shows an instance of the pattern for temporal properties over cardinalities. This example pattern specifies the constraint C4 of Section 6.2, which states that a customer can change the date of his travel only two times and this must happen between 1 and 5 days after the reservation. The designer should specify the resetting of the clock ($x := 0$) in the transition *reserveP* and define the clock constraint ($x > 1day$) on the first *changePack* transition, and ($x < 5day$) on the second *changePack* transition. Between these two transitions, the designer can specify other transitions that correspond to the booking of a new travel package and the reception of the confirmation email.

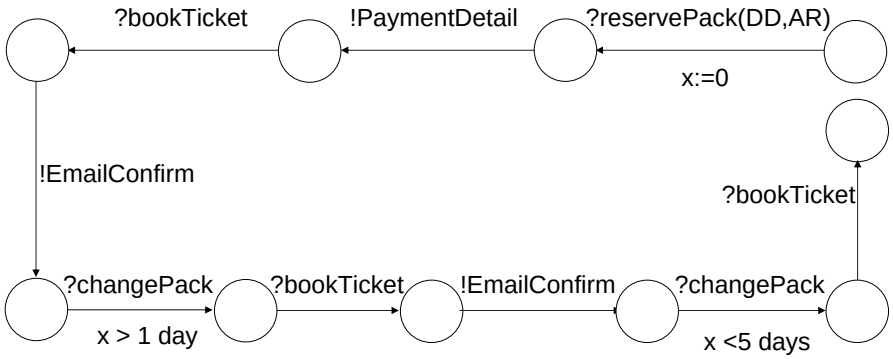

Figure 6.5: Example of temporal property over cardinalities

Pattern for Absolute Time Properties

These properties allow to specify that an activity can be executed only if its current time satisfies some XTUS constraint, which is associated with a transition in the automaton (Figure 6.6). Note that this pattern can be easily combined with the two other patterns discussed above to specify more complex properties.

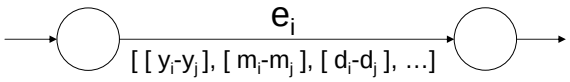

Figure 6.6: Pattern for absolute time properties

As example, we use this pattern for the specification of the constraint C5, which states that a discount is given if the booking is done in a special period (e.g. in the first week of March, April and May). The automaton that defines this constraint would look like the one shown in Figure 6.7 where e_i will be replaced by the transition *bookTicket* and the XTUS expression `[_,[3−5],[1−7]]` is specified on that transition.

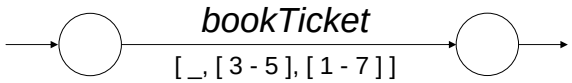

Figure 6.7: Example of absolute time property

The expressive power of timed automata allows to specify most of relative time properties and XTUS allows specifying absolute time properties. As presented below, the designer can easily use XTUS-Automata for specifying temporal properties in Web service applications. He can also combine the proposed specification patterns to express more complex properties.

6.4.4 Verification of Temporal Properties

After specifying the temporal properties related to the Web services composition, the designer can verify the structure and the consistency of his specification.

The designer verifies the syntax and the domain of the XTUS expressions. The verification of the syntax allows to check if the designer follows the XTUS grammar formally defined in Z notation. The designer should use the defined XTUS elements *Element* and take into account the constraints defined in the axiomatic function *def* (cf. Section 6.4.2). The verification of the domain allows to prohibit the designer to specify an incorrect XTUS expression. There is intra- and inter- domain verification. (i) The intra-domain verification allows to verify the domain of elements constituting the XTUS expression: The time and the date, specified by an XTUS expression, should be correct. For example, the domain of the following XTUS expression `[2012,[1−4],[20−30]]` is not correct, as February does not have the days 29 and 30. In addition, the XTUS expression should represent a time or a date in the future. (ii) The inter-domain verification allows to verify the domain of an XTUS expression according to the other ones. Suppose that there are two XTUS expression e_1 and e_2 which are defined respectively on two automata transitions t_1 and t_2. The transition t_2 represents a BPEL activity that will be executed after the activity represented by the transition t_1. The time and the date resulting of e_2 should be greater than the one resulting of e_1. Otherwise, a deadlock state can be reached.

The designer can also formally verify some specific properties using a model checker that supports timed automata, such as UPPAAL [186]. In addition to the graphical modeling of the timed automata and networks of automata, the designer can validate, verify, and simulate the specifications of his Web service compositions. The designer can easily verify the absence of a deadlock state in the timed automata. Based on the temporal logic operators, the designer can also verify safety, liveness and reachability properties [27].

The designer cannot use UPPAAL to verify the properties of his system specified by

XTUS-Automata. Actually, UPPAAL does not support constraints on absolute time, and more specially the XTUS specifications. To resolve this limitation, a formal semantics of XTUS-Automata must be defined using axioms and inference rules. This semantics defines all elements supported by the XTUS language and specifies how to formally combine XTUS within timed automata.

6.5 Aspect-based Enforcement

After the verification phase, to enforce the specified properties we automatically generate aspects from the formal specification. For that purpose, we implemented an XSLT template-based generator engine as an eclipse plug-in. Our plug-in generates enforcement aspects in the AO4BPEL language [44, 45], based on the defined templates and the proposed patterns. Our approach is extensible, as further patterns can be easily defined. The respective transformations to AO4BPEL aspects need to be defined using XSLT templates. These aspects will be deployed together with the processes on an aspect-enabled BPEL engine, which allows the dynamic composition of processes and aspects.

In the following, we first present the aspect language AO4BPEL used for enforcing specified properties. We then explain the aspect generation workflow. After that, we discuss the types of aspects generated for each specification pattern. Finally, we present in detail an aspect template that builds the basis for the aspect generation.

6.5.1 AO4BPEL Language

AO4BPEL [44, 45] is an aspect-oriented extension of the Web service composition language BPEL, which allows for more modular and dynamically adaptable processes.

Like BPEL, AO4BPEL is XML-based. *Aspects* consist of a set of *pointcut-and-advice*. The pointcut language of AO4BPEL uses XPath expressions to select a set of activities using two types of join points. The first type, *activity joint points*, is used to select points corresponding to the execution of activities, e.g., the interception of the invoke or reply activities. The second *internal join points* is used to select points corresponding to internal points during the interpretation of messaging activities, e.g., the interception of the send out of a SOAP message. An *advice* is a BPEL activity that implements some crosscutting logic, e.g., security, monitoring, etc. Special constructs may be used inside the advice to access the input and/or output data of the join points as well as reflective information. Advice can be executed either before, after, or instead of (i.e., around) the intercepted BPEL activities. Further, like BPEL, an AO4BPEL aspect may declare partners, variables, fault handlers, etc. These constructs have the same syntax as in BPEL.

An orchestration engine that supports the dynamic composition of AO4BPEL aspects is available [43]. This engine was implemented on top of IBM's BPEL engine BPWS4J.

6.5.2 Aspect Generation Workflow

The aspect generation workflow is composed of three main phases.

The first phase consists in verifying the conformity the XTUS-Automata specifications with the BPEL process. Certain constraints must be fulfilled by the automata implementing the constraints. The automata states and transitions must match the BPEL process implementing the composition. For example, the name of the transitions in the automaton must match the names of activities in the BPEL process. Further, the name of the input and output parameters of the transitions must match those of the activities. If that phase is not successful, the designer should modify his specifications.

The second phase consists in identifying all instances of the specification patterns in the XTUS-automata specifications describing the system and its temporal constraints. Note that, the designer can separately specify his specifications as a set of XTUS-Automata specifications that instantiate the specification patterns.

The third phase consists in automatically generate AO4BPEL aspects by filling the placeholders and parameters of the respective aspect templates with data from the concrete pattern instances. We proposed the specifications patterns as incomplete and parameterized AO4BPEL aspects that enforce the properties expressed by the corresponding pattern. An aspect template contains generic enforcement logic that is independent from a particular scenario. Certain data and parameters need to be completed from the formal specification, and this is done using the following two mappings:

- Each specified automaton transition is mapped to an AO4BPEL pointcut as an XPath-expression [193] that matches WS-BPEL activities, i.e., in order to know when to reset or test a counter.

- Each temporal constraint from the automaton model is mapped to the expression language of WS-BPEL which is evaluated by a time helper service. The expression contains the concrete clock name and clock values.

Once the template is instantiated, the generated aspects can be deployed together with the process on the AO4BPEL engine. Thus, they ensure through runtime monitoring the enforcement of the respective constraint.

6.5.3 The Runtime Enforcement Aspects

We generate two types of aspects to enforce temporal properties specified as instances of the pre-defined specification patterns. The first type uses an after advice and allows to start a timer (*timer aspect*) or an activity counter (*counter aspect*) after the execution of a given activity. The second type uses an around advice (*decision aspect*) and allows or prohibits

the execution of the intercepted activity according to the specified constraint and to the state of the timer and/or the counter defined in the first aspect.

Next, we explain in more detail which aspects are needed for each specification pattern, the mapping of constructs of the specification XTUS automaton to AO4BPEL pointcuts and advice, and the advice logic in each case.

Aspects for Duration Properties

The enforcement of duration properties requires a *timer aspect* and a *decision aspect*. The pointcut of the timer aspect intercepts the BPEL activity specified as a reset transition (i.e. the automaton transition labeled with a reset of the clock variable or the transition labeled with the definition of a clock as input or output parameters). The advice of this aspect invokes a helper time Web service to start a timer after the execution of the intercepted activity. For the decision aspect, the pointcut intercepts the execution of the activity specified as an automaton transition labeled with a temporal constraint. The advice of this aspect contains two parts. In the first part, a helper Web service is invoked to stop the timer before the execution of the intercepted activity. The identification of the timer in the two aspects is ensured by the name of the clock variable used in the two transitions specifying the intercepted activities. In the second part, the aspect checks the constraints defined in the automaton transition that corresponds to the intercepted activity. If the constraint is satisfied, the activity will be executed. Otherwise, the aspect prohibits the activity and an exception will be raised.

Aspects for Temporal Properties over Cardinalities

The enforcement of temporal properties over cardinalities can be seen as an extension of the monitoring of duration properties. In addition to the timer aspect, a counter aspect is generated, which allows to calculate the number of times an activity is executed. The pointcut of the counter aspect intercepts the BPEL activity specified by a transition that is repeated more than one time and defined between a reset transition and a transition that is labeled with a temporal constraint. The advice invokes a helper Web service to start and increment the counter. The decision aspect verifies also the number of times the activity is executed against the number of the transition labeled with activity name.

Aspects for Absolute Time Properties

The monitoring of the absolute time pattern requires only a decision aspect. The generated pointcut intercepts the automaton transition labeled with an XTUS expression. The advice verifies if the current time and date satisfy the XTUS constraints as formally defined in

Section 6.4.2. Based on that the advice allows or prohibits the execution of the intercepted activity.

6.5.4 Aspect Templates

In the following, we present in detail an aspect template and illustrate the aspect generation based on the concrete examples shown in Section 6.4.3.

Listing 6.1 shows the template of the timer aspects (lines 16–39) and the decision aspects (lines 45–76) defined in Section 6.5.3. As an example, we illustrate the use of this template to generate monitoring aspects for the duration patterns defined in Figure 6.3. The template uses loop constructs to iterate over the set of transitions in the XTUS-Automaton specification (lines 14 and 41, 43 and 78) and defines several placeholders (lines 18, 29, 47, 55, and 67).

For the timer aspect, the pointcut (lines 17–19) intercepts the corresponding BPEL activity specified by the reset transition. The *Reset Transition* placeholder (line 18) is replaced by a pointcut that matches the process activity that leads to the resetting of the corresponding counter. In an AO4BPEL aspect, the pointcut representing the interception of an activity is defined as an XPath-expression in the form of
`//process//`*`activityType`*`[@operation="`*`operationName`*`"]`.
For the example in Figure 6.3, the timer aspect intercepts the execution of the activity *reservePack* which resets the counter x. The *Reset Transition* placeholder is replaced by `//process//reply[@operation="reservePack"]`. After the execution of the intercepted activity, the advice (lines 21–38) invokes the operation *startTimer* (lines 4 and 34) on a helper time Web service to measure the execution time of the intercepted activity. This operation takes the name of the monitored activity and the clock variable as input parameters. The name of the activity is set by an assign activity (lines 23–32) to the name of the current join point activity, which is accessed by using the reflective AO4BPEL variable ThisJPActivity (line 25). The *clock name* placeholder (line 29) is extracted from the automaton transition corresponding to the intercepted activity.

For the monitoring of temporal properties over cardinalities (Figure 6.4), an additional counter aspect should be generated with the same structure as the timer aspect. This aspect intercepts the corresponding activity and invokes another helper Web service to calculate the number of times the intercepted activity has been executed so far.

The decision aspect (lines 45–76) is defined with an around advice. The pointcut (lines 46–48) intercepts the activity modeled by the automaton transition labeled with the temporal constraint. The *Enforce Transition* placeholder (line 47) is replaced by a pointcut that matches the corresponding activity. For the example shown in Figure 6.3, the pointcut `//process//reply[@operation="bookTicket"]` matches the activity representing the ticket booking, which should be performed in the first 30 minutes after reservation. In the

advice (lines 50–75), a helper Web service is invoked to count the number of times the intercepted activity is repeated (line 60) and calculate the duration between this activity and the activity intercepted in the first aspect (line 63). At the end, the temporal constraint is verified using a switch condition (lines 66–73). The *Constraint Expression* placeholder (line 67) is replaced by the technical expression in the expression language of WS-BPEL. E.g., in Figure 6.3 the temporal constraint $x < 30$ defined on the *bookTicket* transition is mapped to "`getVariableData(timeWsResponse,duration) > 30`". If the constraint is satisfied, the corresponding activity will be executed using the keyword *proceed* (line 68), otherwise the activity will not be executed and an exception will be raised (line 70–72).

```
<aspect>

  <partners>
    <partner name="TimeWService" serviceLinkType="tns:TimeWServiceSLT"/>
    ...
  </partners>

  <variables>
    <variable name="timeWsRequest" messageType="tns:TimerRequest" />
    <variable name="timeWsResponse" messageType="tns:TimerResponse" />
    ...
  </variables>

  BEGIN-FOR-EACH (clock reset transition)

  <pointcut-and-advice name="startTimer">
    <pointcut name="startTimePc" contextCollection="true">
      Reset Transition
    </pointcut>

    <advice type="after">
      <sequence>
        <assign>
            <copy>
                <from variable="ThisJPActivity" part="name" />
                <to variable="timeWsRequest" part="activityName" />
            </copy>
            <copy>
                <from expression="'clock name'" />
                <to variable="timeWsRequest" part="identifier" />
            </copy>
        </assign>

        <invoke name="invokeStart" partner="TimeWService" operation="startTimer"
                inputVariable="timeWsRequest" .../>

      </sequence>
    </advice>
  </pointcut-and-advice>

  END-OF-FOR-EACH

  BEGIN-FOR-EACH (constrained transition)

  <pointcut-and-advice name="TimeMonitor">
    <pointcut name="VerificationTimePC" contextCollection="true">
      Enforce Transition
    </pointcut>

    <advice type="around">
      <sequence>
        <assign>
            ...
```

```
54            <copy>
55               <from expression="' clock name '" />
56               <to variable="timeWsRequest" part="identifier" />
57            </copy>
58         </assign>
59
60         <invoke name="invokeStop" partner="TimeWService" operation="stopTimer"
61                inputVariable="timeWsRequest" .../>
62
63         <invoke name="invokeDuration" partner="TimeWService" operation="getDuration"
64                outputVariable="timeWsResponse" ... />
65
66         <switch name= "ProhibitAllowAction">
67           <case condition = " Constraint Expression ">
68             <proceed />
69           </case>
70           <otherwise>
71             ...
72           </otherwise>
73         </switch>
74       </sequence>
75     </advice>
76   </pointcut-and-advice>
77
78   END-OF-FOR-EACH
79
80 </aspect>
```

Listing 6.1: Template of the generated AO4BPEL Aspect

6.6 Overview of the Prototype

We implemented a research prototype to prove the concepts of the proposed approach for implementing temporal properties in Web service composition. The prototype is an Eclipse plug-in which allows the specification of these properties and the generation of the corresponding enforcement code.

We developed an Eclipse visual editor based Eclipse Modeling Framework [64] (EMF), Graphical Editing Framework [77] (GEF) and Graphical Modeling Framework [80] (GMF) frameworks. Using this editor, the designer models the XTUS-Automata presenting the specification of the temporal properties of his Web service composition. Four steps were required to create this editor.

1. We specified the meta-model of the XTUS-Automata language presenting as an *ecore* file. This meta-model is based on EMF and contains the meta-classes required for drawing XTUS-Automata.

2. We specified a graphical representation of each defined meta-class. This representation extends the standard representation of timed automata (e.g., state as a circle, transition as a link between circle, etc.) by a label on a transition for the XTUS expression.

3. We developed a palette that includes the required buttons for drawing the elements constituting the XTUS-Automata (i.e., automata state, transition, initialization, guard, and XTUS expression). To easily use this palette, we defined colors, icons, and groups of buttons.

4. We defined a mapping between the meta-model (ecore file), the graphical representation and the palette. After that, we generated, using the EMF and GMF frameworks, Java code to finalize the creation of the Eclipse plug-in.

A screenshot of our eclipse visual editor is shown in Figure 6.8.

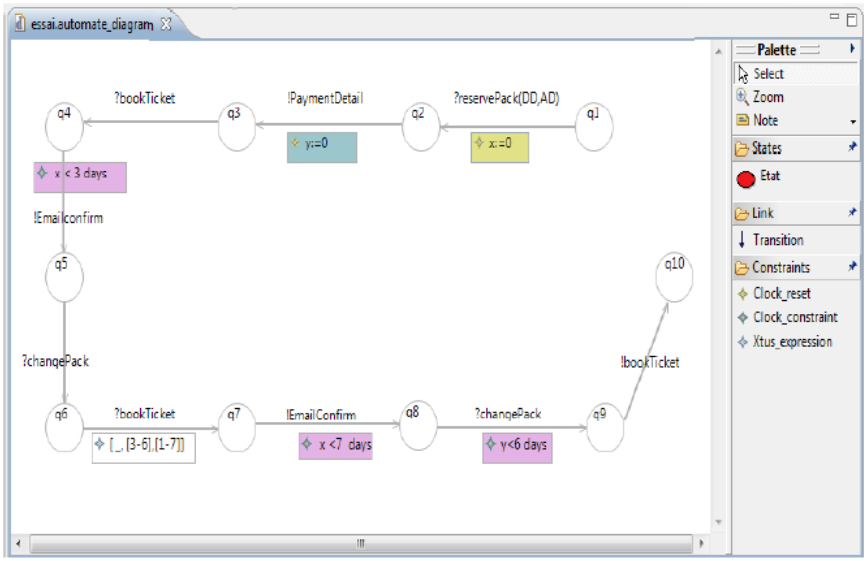

Figure 6.8: Screenshot of the XTUS-Automata-to-AO4BPEL plug-in

After specifying the temporal properties of the Web service composition, the designer should verify the structure of his specification as previously discussed in Section 6.4.4. Our plug-in automatically translates the graphical representation of the temporal properties into an XML-based textual representation. We developed a module, as a part of this plug-in, which parses and extracts the required information from the XMI files to verify the structure

of the specification. Using the menu item *verify properties* shown in the Figure 6.9, the designer verifies:

- The conformity of the specification with the BPEL process: for example, the name of the automata transitions are conform with the defined BPEL activities.
- The static semantics of the automata guard: for example, the used variables should be already initialized in a previous transition.
- The syntax and domain of the XTUS expression as detailed in the verification phase (cf. Section 6.4.4).

The designer can also verify other properties of the specification using the model checker UPPAAL. Using the menu item *export to UPPAAL* shown in Figure 6.9, the designer can export his specification designed in our plug-in into a XML file that can be consumed by the UPPAAL model checker.

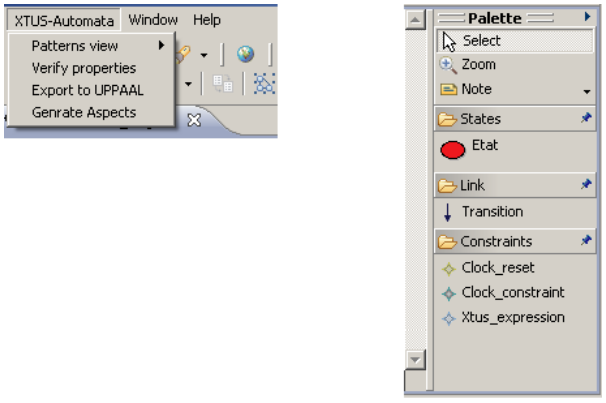

Figure 6.9: Menus of the XTUS-Automata-to-AO4BPEL plug-in

If the verification step is successful, the designer can easily identify all instances of the specification patterns in the XTUS-Automaton (or automata) describing his composition system and its temporal properties. Using the item *view patterns*, the designer verifies that the specification is correct (i.e., based on the proposed patterns) and extracts the required information for each instance to generate the corresponding aspects. The identification of the instance step is performed by parsing the XMI file representing the specification of the temporal properties.

Using the menu item *Code generation*, the designer can automatically generate the aspect code, according to the defined templates, from each identified pattern instance. At

runtime, the generated AO4BPEL aspects will be woven in the BPEL process (without stopping the process) using the AO4BPEL engine [43].

6.7 Related Work

In this section, we present related work on the formal specification and the implementation of temporal properties in Web service compositions. Several formal methods were used to model Web service compositions, such as Petri nets [86], and finite automata [72]. The main goal of these works is to verify the business process properties using model checking tools. The majority of these works do not consider the implementation of non-functional properties such as temporal properties or quality of service.

Other works [164, 177] used different languages and tools to specify and monitor service level agreements (SLA). Most of these works describe in a high-level specification the contract between the customer and the service provider in terms of quality of service properties. These works focus on the description of software qualities such as maintainability, robustness, performance. They just verify, at runtime, if the these properties (defined as a contract) are satisfied. They do not stop/skip the execution of a critical operation if one of these properties is not satisfied. In addition, these works do not address temporal properties required in Web service composition.

Another group of related work concentrates on the specification of non-functional properties, and more specifically on temporal properties. In [112], the authors propose a new model, called web service timed state transition systems, inspired from timed automata for specifying temporal behavior of web services compositions. The authors use also interval temporal logic to express complex temporal requirements. Based on this model, they propose a model checking approach for temporal properties on top of BPEL. Similar to our approach, the authors extend timed automata to cover more temporal properties. The semantics of this language is not formally defined. This language does not support the specification of absolute temporal properties defined in our approach using the XTUS language. Another limitation of this work is that it does not propose any mechanism to control at runtime the specified temporal properties.

Baresi et al. [18] propose a temporal extension of an XML-based assertion language called Timed WSCoL, which allows expressing temporal properties over the actions that occur during the process execution. The authors propose powerful operators (such as *between* and *within*) in addition to the linear temporal operators (such as *always* and *until*). A semantics of this language is formally defined. For the monitoring phase, they propose to use Dynamo, an aspect-oriented extension for ActiveBPEL, as well as an external analyzer to ensure asynchronous monitoring. Compared to our approach, Timed WSCoL is not really expressive and does not cover all temporal properties addressed in our work, especially the

properties defined by XTUS. Also, this work does not provide any mechanism to formally verify the specified properties.

The approach proposed by Barbon et al. [17] is quite similar to ours. The authors propose the specification of temporal properties and statistic properties in BPEL processes using the RTML language (Runtime Monitoring Specification Language). They also generate Java code for monitoring the specified properties. However, compared to our approach, this work is less expressive than ours, as absolute time properties for example are not supported. Further, the monitoring code generated is not modular as it is mixed with application code. In addition, that approach is static, i.e., it does not allow dynamic changes of temporal properties at runtime, which is possible in our approach thanks to dynamic weaving in the AO4BPEL engine.

In [84], Guermouche et al. specify temporal properties using an extension of the WSTTS language (i.e. Web Service Timed Transition Systems). They use temporal properties to analyze the compatibility in Web service composition. Compared to to our approach, the authors use a similar model to specify temporal properties in Web service composition, they propose also an extension of timed automata. However, their extension to support absolute temporal properties is not formally defined: no formal semantics is provided. They propose just to add an interval of date under the automaton transitions. In addition, in [84], no monitoring mechanism is presented to verify at runtime the specified properties.

6.8 Conclusion

The contributions presented in this chapter are many-fold: First, we defined a new formal language, called *XTUS-Automata* that combines *timed automata* and *extended time unit system*. This language allows specifying temporal properties involving both *relative time* and *absolute time*. Second, we provided generic *specification patterns* that ease the specification of temporal properties in Web services compositions. Third, we designed and implemented an aspect-based monitoring mechanism, in which formally specified temporal constraints are translated automatically to modular aspect code in the aspect-oriented workflow language AO4BPEL. As AO4BPEL supports dynamic weaving the aspects can be integrated at runtime with the processes. The generated aspects intercept the execution of process activities. If the respective temporal properties are satisfied, the activity will be executed, otherwise the aspect prohibits the activity execution and throws an exception.

Our approach suffers from some limitations and usage restrictions: The proposed specification patterns can only specify restricted set of temporal properties. As a solution, we plan to extend our approach by other specification patterns for more complex temporal properties and will implement the necessary templates for generating the monitoring aspects. On the other hand, no mechanism is proposed to formally verify if there is a inconsistency

between timed automata and XTUS constraints. Software testing [41] can be applied to resolve such limitation. Finally, the AO4BPEL engine is available only for BPEL1.0.

Chapter 7
Conclusions

7.1 Summary

In this thesis, we proposed a solution for implementing non-functional safety properties in distributed systems in a reliable and modular way. It consists of a holistic approach for the formal specification and the aspect-based enforcement of these properties.

Before discussing the motivations of this work, this thesis presented the necessary background comprising brief definitions and reviews of the mechanisms and methods used in this PhD work. In addition, a comparative study on existing approaches for specifying non-functional properties was presented. These approaches use different specification mechanisms such as UML profiles, Aspect-oriented modeling, and formal methods and languages. A detailed study on runtime verification approaches was also presented. This study focuses on existing approaches that automatically generate code from a high-level specification and use Aspect-oriented programming as a runtime verification mechanism.

To show the usefulness of our approach, we detailed the limitations of existing research works. These limitations are due to the use of semi-formal methods at the specification level, the problem of crosscutting concerns modularity, and the gap between specification and implementation.

The central topic of this work is the combination of formal methods and aspect-oriented programming for specifying and enforcing non-functional safety properties. The specification of non-functional safety properties is formally verified and its structure is validated according to a well defined meta-model. Then, it is automatically translated to aspect code. Our approach, called Seven-pro, solves the problems and limitations identified in existing approaches.

After detailing the proposed approach and its phases, this thesis presented how Seven-pro can cover several types of non-functional properties and can also be applied in different application domains and to different types of architectures. Our approach was applied to three case studies presented in this thesis.

The first case study deals with dynamic software architecture of object-oriented applications. SEVEN-pro is applied for specifying and enforcing architectural invariants. A combination of the Z notation and Petri nets is proposed for specifying such properties. These properties are classified into: properties on objects and object cardinality, properties on object relationships and their cardinality, properties on methods that modify the software architecture, and method call protocols. These properties are formally verified using the Z/EVES theorem prover and automatically translated to AspectJ aspects.

The second case study consists of applying SEVEN-pro to specifying and enforcing access control policies. *Temporal$\mathcal{Z}$*, as a formal language, is extended by specific security predicates for specifying separation-of-duties and delegation policies on top of role-based access control. ALPHA, as an expressive aspect-oriented language, is extended also by a security library for supporting the specified policies. An automatic translation process is defined from *Temporal$\mathcal{Z}$* specifications as a domain-specific specification language to ALPHA code as a domain-specific implementation language.

The third case study focuses on the specification and the enforcement of temporal properties in Web service composition. The formal language XTUS-Automata is defined for specifying relative and absolute time related properties. This language is a combination of the XTUS language with timed automata. The specification of temporal properties is automatically translated to AO4BPEL aspects which can be woven in a modular way in the BPEL process.

7.2 Review of the Seven-Pro Approach

In this section, we present a review of our approach. We will focus on the most important criteria for reviewing methods and tools. The considered criteria are inspired from [117, 143, 169] such as generality, usability, and applicability. For each criterion, we present a general definition, its proof and some limitations.

7.2.1 Reviewing Strategy

Our review follows a strategy based on three classes of criteria. The first class includes review criteria that can be addressed with different case studies and examples [117]. These case studies should be selected using an appropriate process in order to cover the different concepts and to express the advantages and the disadvantages of the reviewed method. The second class concerns criteria based on logical reasoning. This class covers the criteria that can be deduced from the tools and facilities provided with the method. Criteria belonging to the third class are addressed using empirical studies which reveal the difficulties and research challenges in the practical use of the approach and its software. The empirical studies allow answering important questions about, for example, the real user benefits, the

required time and effort, etc. For lack of time and the complexity of the empirical study process, we do not consider this class of evaluation methods in this work. Instead, we will focus on the other two classes.

7.2.2 Review based on the Analysis of Case Studies

In the following, we focus on the generality and simplicity criteria.

The *generality* criterion states that the proposed method can cover most important aspects of the application domain. In order to review the generality of our approach, we apply the SEVEN-pro approach to three types of properties. First, we applied our approach to software architectural invariants as an example of structural properties. Second, we applied it to access control policies as an example of qualitative behavioral properties. Third, we applied our approach to temporal properties in Web service compositions as an example of quantitative behavioral properties. The details of these applications are presented in the three previous chapters.

The *simplicity* criterion expresses that the desired application can be easily defined based on the proposed approach.

Despite the difficulty and the diversity of the considered non-functional properties (architectural properties, security properties, temporal properties), we easily applied this approach in three complex case studies. This is due to the fact that the approach facilitates the design work in the different phases of development. SEVEN-pro minimizes the difficulty of using formal methods and mechanisms by proposing useful and powerful specification patterns. A non-expert designer can easily specify his non-functional properties. He just needs to instantiate the provided patterns to create his specification. In the verification phase, the use of XML and XML schema allow to easily detect if the specification corresponds to the predefined patterns or not. In the implementation phase, the executable aspect code is automatically generated while assuming that the functional code of the application conforms to the formal specification. The designer should just deploy the generated aspects.

Our approach has significant advantages but also some limitations with respect to the simplicity criterion. Despite the advantage of using specification patterns in the first phase of our approach, which facilitates the tasks of designers and minimizes the complexity of the formal methods, the use of graphical notations, such as UML, is the most simple way. We can even take advantage of design patterns in the UML language. To remedy the problems related to this language such as the lack of a rigorous semantics and the verification techniques, we propose to generate formal specifications from a graphical modeling of non-functional properties. We have already applied this approach to specifying and enforcing

architectural invariants in dynamic software architecture [110,111]. We combine the SEVEN-pro approach with one that proposes a UML profile for modeling the dynamic architecture and its constraints. In addition, we automatically generate a Z specification from the model.

7.2.3 Review based on Logical Reasoning

In this section, we present the criteria belonging to the class describing the review based on logical reasoning. We focus on usability, applicability, and code quality.

The *usability* criterion expresses that the developed application should be easy to use by the class of users for whom it was intended [143]. We explain how the SEVEN-pro approach simplifies the work of the designers in two steps.

First, we note that we developed an Eclipse plug-in supporting the proposed approach for each case study. The details of each plug-in and their functionalities are presented in the previous chapters. The accessibility of the different functionalities of the plug-ins is ensured using menu-items. For example, we consider the developed plug-in for specifying and enforcing temporal properties of Web service composition. This plug-in supports

- the graphical modeling of temporal properties using the extended timed automata,
- the translation of the graphical model to XMI files to verify the conformity to the patterns,
- the generation of the format supported by the model checker UPPAAL for the formal verification, and
- the generation of the corresponding enforcement aspects in AO4BPEL.

Second, we detail the benefits of the automatic transition between the different phases of the development process for implementing non-functional properties. This automatic transition saves time and effort of the designers.

Figure 7.1 presents approximately the benefit (color gradient) in terms of effort and time provided to the designer using the SEVEN-pro approach and its plug-ins. SEVEN-pro automates more than 50% (5/8). In the specification step (1), the benefits are achieved through the provided patterns: the designer should just instantiate the patterns to create the specification. The validation step (2) is performed automatically. An XML file representing the specification is automatically generated and validated according to the XML schema that represents the patterns. In the enforcement phase, the aspect code is also automatically generated and integrated into the functional code of the application. This is achieved by the proposed templates and automatic generation tools.

As we presented above our approach is easy to use by the software designer because it automatically generates aspect code from the high level specification. In this context, the

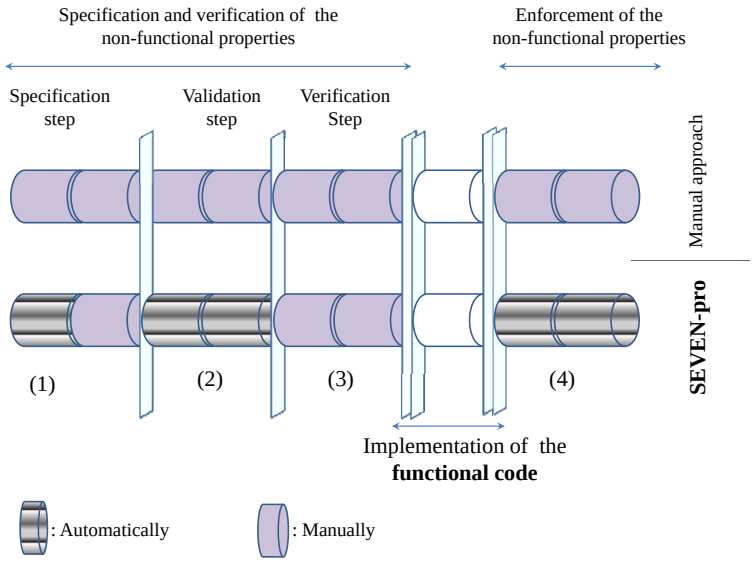

Figure 7.1: Benefits of Seven-pro approach in terms of effort and time

limitation that is still not resolved is how to prove that the generation of aspects is correct, i.e., how to confirm that the generated aspects conform to the formal specifications.

The *applicability criterion* expresses that the method meets the needs of the designer and covers the development phases [143].

Generally, for developing non-functional properties, the designer needs powerful mechanisms to quickly express these properties and concentrate on the functional part of the application.

The designer needs an approach to easily specify the non-functional properties. He also requires mechanisms to validate and verify these properties. In addition, the designer has difficulties in refining his specification into a corresponding implementation. These needs are supported by our SEVEN-pro approach and its associated tools. We propose, in the SEVEN-pro approach, a high-level specification of the non-functional properties based on some specification patterns and followed by the validation and verification steps. Finally, the verified specifications are automatically translated to aspect code and woven with the functional part of the application.

The *quality of the code* is an important review criterion for non-functional properties. Based on Aspect-Oriented Programming, the generated code for enforcing these properties is well modularized and separated from the functional code. If the invariants change, e.g., to

accommodate some changes in the requirements, the designer does not need to localize the code affected by the implementation of the invariants enforcement logic. He can just modify the specification and generate the corresponding aspects. The designer can easily modify or define a new property since the corresponding code is automatically generated in a modular way. In addition, aspect-oriented techniques allow targeting faster aspect compilation and runtime efficiency. Compared to writing aspects manually, the generation of aspect code from a formally consistent specification avoids the risk of overlap between the properties and between the functional application code and the code implementing non-functional properties.

7.3 Future Work

In this section, we present some directions for future work. We focus on three areas: the extension of the SEVEN-pro approach, the application of the SEVEN-pro approach to other properties, and the validation of aspect generators.

1. Works on the SEVEN-pro approach

   - *Extend* SEVEN-*pro by a high-level modeling phase*
     The use of formal methods in software development has several advantages such as reliability, safety, and correctness. However, it requires a high degree of mathematical expertise and a solid foundation in logic, which is a difficult task for non-expert designers. For overcoming this limitation in our approach, we plan to extend the SEVEN-pro approach with a high-level modeling phase.

     In the modeling phase, the designer models his non-functional properties using a semi-formal language based on graphical notations. Since UML is the most appropriate and adopted language for graphically modeling such properties, we plan to define a UML profile for each type of properties. We can define design patterns to help the designer for modeling the generic properties. We will propose also a generation process which automatically translates UML models into the corresponding formal specification in order to formally verify its consistency.

   - *Extend* SEVEN-*pro to support decentralized distributed applications*
     Although our approach can be applied to object-oriented, component-based, service-oriented architectures, it provides centralized solutions to control and monitor distributed applications. This limitation comes from the fact that the generated aspects are not distributed—they are localized on the host server (a centralized host). The advices of these generated aspects can be executed only on the host server on which the pointcut is matched.

For supporting decentralized monitoring, we plan to automatically translate formal specifications to AWED aspects [149, 185], as distributed aspects. The pointcuts of these aspects are more expressive than the traditional pointcut language by supporting remote sequence. The advice of AWED aspects can also be executed on any hosts different from the host where the corresponding pointcut is matched.

2. Works on applying SEVEN-pro to other properties and application domains

   - *Applying* SEVEN*-pro to real-time embedded software systems*
     We plan to apply the SEVEN-pro approach to specifying and enforcing resource-related properties of real-time embedded software systems. These systems evolve during the execution according to the environment context, user requirements and resources limitations [121].

     In this context, the designer formally specifies his system and it evolution by defining the reconfiguration activities (i.e. creating/deleting, activating/deactivating threads) associated with their pre-conditions. The latters check the availability of resources before any reconfiguration activity (e.g., CPU utilization, memory size, bandwidth, etc.). Extended timed Petri nets that support the specification of resources can be used as a formal specification language. These preconditions will be automatically translated into aspects which verify at runtime if the resource-related constraints are satisfied.

   - *Apply* SEVEN*-pro on other access control policies*
     The development of secure applications is usually a complex task for developers. No errors should occur during the development process. These policies are often changed to manage the system and user requirements. Our approach is well-suited for implementing access control policies.

     The designer can easily obtain a secure application by formally specifying his security policy. The code that enforces the specified constraints is generated automatically and integrated modularly within the functional code. In addition to the RBAC and its extensions, other access control policies can also be applied such as Chinese Wall [38], OrBAC (Organization Based Access Control) [103], TRBAC (Temporal Role-Based Access Control Model) [28], etc. Each policy requires special features in the specification language. Different aspect languages can also be used depending on applications domains and architectures.

3. Works on the validation of aspect generators
   To fully benefit from the use of formal methods in our approach in terms of verification of properties, the correctness of aspect code generation has to be verified formally.

This verification of the correctness of our aspect code generator is similar to the validation of a compiler. The generated aspect code should have the same semantics as the formal specification. There are three directions for resolving this problem.

- The first direction is the use of formal methods such model checking, static analysis, theorem proving, etc. These verification techniques can be applied to the generated aspect code. The generator is only a weak connection between the formal specification and the generated code.
- The second direction is the also about the use of formal methods on the code generator itself. The syntax and the semantics of all constructs of the source language (formal language) and also of the output language (aspect language) should be formally defined. In addition, a correspondence between these two semantics implemented as transformation rules should be defined. The verification of the correctness of the generator consists in ensuring that the transformation rules are correctly implemented.
- The third direction is about the use of software testing for detecting the errors and bugs of the generator. A large number of possible execution scenarios are automatically generated to test if the generated aspect code is conform to the formal specification by comparing the behavior of generated advice to the expected behavior.

Bibliography

[1] Ali E. Abdallah and Etienne J. Khayat. Formal Z Specifications of Several Flat Role-Based Access Control Models. In *Proceedings of the 30th Software Engineering Workshop (SEW)*, pages 282–292. IEEE, 2006.

[2] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, 1995.

[3] Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, editors. *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, volume 2001 of *LNCS*. Springer, 2001.

[4] Nazareno Aguirre and Tom Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE)*, pages 271–274. IEEE Computer Society, 2002.

[5] Gail-Joon Ahn. Specification and Classification of Role-based Authorization Policies. In *Proceedings of the 12th International Workshop on Enabling Technologies (WETICE)*, page 202. IEEE Computer Society, 2003.

[6] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the 4th ACM workshop on Role-based Access Control*, pages 43–54. ACM, 1999.

[7] Gail-Joon Ahn and Ravi Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, 3(4):207–226, 2000.

[8] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 187–197. ACM, 2002.

[9] Noorazean Mohd Ali and Awais Rashid. A State-based Join Point Model for AOP. In *Proceedings of the 1st Workshop on Views, Aspects and Role (VAR)*, 2005.

[10] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the 20th Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM, 2005.

[11] Houssem Aloulou, Monia Loulou, Slim Kallel, and Ahmed Hadj Kacem. RDyMASS: Reliable and Dynamic Enforcement of Security Policies for Mobile Agent Systems. In *Proceedings of the 2nd International Workshop on Autonomous and Spontaneous Security (SETOP)*, volume 5939 of *LNCS*, pages 237–252. Springer, 2009.

[12] Rajeev Alur and David L. Dill. A theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[13] Hasan Amjad. Combining Model Checking and Theorem Proving. Technical report, University of Cambridge, Computer Laboratory, 2004.

[14] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Transactions Aspect-Oriented Software Development I*, pages 135–173, 2006.

[15] Rob Arthan. ProofPower. www.lemma - one.com/ProofPower/index, 2009.

[16] Anaheed Ayoub, Ayman M. Wahba, Ashraf M. Salem, and Mohamed A. Sheirah. Code Synthesis for Timed Automata: A Comparison Using Case Study. In *Proceedings of the 2nd International Conference of the Abstract State Machines, Alloy, B and Z (ABZ)*, volume 5977 of *LNCS*, page 403. Springer, 2010.

[17] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of the 4th IEEE International Conference on Web Services (ICWS)*, pages 63–71. IEEE Computer Society, 2006.

[18] Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. A Timed Extension of WSCoL. In *Proceedings of the 5th IEEE International Conference on Web Services (ICWS)*, pages 663–670. IEEE Compuer Society, 2007.

[19] Ezedin Barka and Alaa Aly. Implementation of Role-Based Delegation Model/Flat Roles (RBDM0). In *Proceedings of the 6th Annual UAE University Research Conference*, 2006.

[20] Ezedin Barka and Ravi Sandhu. A Role-based Delegation Model and Some Extensions. In *Proceedings of 23rd National Information Systems Security Conference (NISSC)*, 2000.

[21] Ezedin Barka and Ravi Sandhu. Role-based Delegation Model/Hierarchical Roles (RBDM1). In *Proceedings of the 20th Annual Conference on Computer Security Applications (CSAC)*, pages 396 – 404. IEEE, 2004.

[22] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jorgen Fischer Nilsson. Synthesis of Programs in Computational Logic. In *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, volume 3049 of *LNCS*, pages 30–65. Springer, 2004.

[23] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[24] David Basin, Ernst-Ruediger Olderog, and Paul E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security (ASIACCS)*, pages 70–81. ACM, 2007.

[25] Thais Batista, Antônio T. Gomes, Geoff Coulson, Christina Chavez, and Alessandro Garcia. On the Interplay of Aspects and Dynamic Reconfiguration in a Specification-to-Deployment Environment. In *Proceedings of the 2nd European conference on Software Architecture (ECSA)*, volume 5292 of *LNCS*, pages 314–317. Springer, 2008.

[26] Lujo Bauer, Jay Ligatti, and David Walker. Composing Security Policies with Polymer. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314. ACM, 2005.

[27] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185 of *LNCS*, pages 200–237. Springer, 2004.

[28] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A Temporal Role-based Access Control Model. *ACM Transactions on Information and System Security*, 4(3):191–233, 2001.

[29] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of Specification Patterns with the B Method. *CoRR*, abs/cs/0610097, 2006.

[30] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *Proceedings of the 21st Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 109–124. ACM, 2006.

[31] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient Control flow Quantification. In *Proceedings of the 21st Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 125–138. ACM, 2006.

[32] Eric Bodden. J-LO - A Tool for Runtime-Checking Temporal Assertions. Diploma thesis, RWTH Aachen University, 2005.

[33] Eric Bodden. J-LO: Java Logical Observer. `http://www-i2.informatik.rwth-aachen.de/Forschung/RV/JLO/`, 2005.

[34] Eric Bodden. *Verifying Finite-state Properties of Large-scale Programs*. PhD thesis, McGill University, 2009.

[35] Maroua Bouzid and Antoni Ligeza. Algebraic Temporal Specifications with Extended TUS. Hierarchical Granular Terms and Their Applications. In *Proceedings of the 17th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 249–253. IEEE, 2005.

[36] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Publishing, 2003.

[37] Chiara Braghin, Daniele Gorla, and Vladimiro Sassone. A Distributed Calculus for Role-Based Access Control. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations (CSFW)*, pages 48–60. IEEE Computer Society, 2004.

[38] David F. C. Brewer and Michael J. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 206–214. IEEE Computer Society, 1989.

[39] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. HOL-Z 3.0. http://www.brucker.ch/projects/hol-z/, 2008.

[40] Robert Büssow and Wolfgang Grieskamp. Combinig Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In *Proceedings of the 3rd Asian Computing Science Conference on Advances in Computing Science (Asian)*, volume 1345 of *LNCS*, pages 46–56. Springer, 1997.

[41] Ana Cavalli and Sudipto Ghosh, editors. *Proocedings of the IEEE International Conference on Software Testing Verification and Validation (ICST)*. IEEE, 2010.

[42] Çağdaş Cirit and Feza Buzluca. A UML Profile for Role-based Access Control. In *Proceedings of the 2nd International Conference on Security of Information and Networks (SIN)*, pages 83–92. ACM, 2009.

[43] Anis Charfi. AO4BPEL. `http://www.stg.tu-darmstadt.de/research`, 2007.

[44] Anis Charfi. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD Thesis, TU Darmstadt, Germany, 2007.

[45] Anis Charfi and Mira Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web Journal*, 10(3):309–344, 2007.

[46] Feng Chen, Marcelo d'Amorim, and Grigore Rosu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 357–372. Springer, 2004.

[47] Feng Chen and Grigore Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[48] Feng Chen and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 546–550. Springer, 2005.

[49] Feng Chen and Grigore Rosu. MOP: an Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 569–588. ACM, 2007.

[50] Kung Chen. Using Dynamic Aspects for Delegating Fine-Grained Access Rights. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC)*, pages 783–789. IEEE Computer Society, 2005.

[51] Kung Chen and Ching-Wei Lin. An Aspect-Oriented Approach to Declarative Access Control for Web Applications. In *Proceedings of the 8th Asia Pacific Web Conference (APWeb)*, volume 3841 of *LNCS*, pages 176–188. Springer, 2006.

[52] Edmund M. Clarke, Orna Grumberg, and David E. Long. Verification Tools for Finite-State Concurrent Systems. In *Proceedings of REX School/Symposium on A Decade*

of Concurrency, Reflections and Perspectives, volume 803 of *LNCS*, pages 124–175. Springer, 1994.

[53] Kendra Cooper, Lirong Dai, Sergiu Dascalu, Nehal Mehta, and Sujala Velagapudi. Towards Aspect-oriented Model-driven Code Generation in the Formal Design Analysis Framework. In *Proceedings of the 2007 International Conference on Software Engineering Research & Practice (SERP)*, pages 628–633. CSREA Press, 2007.

[54] Jason Crampton and Hemanth Khambhammettu. Delegation in Role-Based Access Control. In *Proceedings of the 11th European Symposium on Research in Computer Security(ESORICS)*, volume 4189 of *LNCS*, pages 174–191. Springer, 2006.

[55] Lirong Dai. *Formal Design Analysis Framework: An Aspect-oriented Architectural Framework*. PhD thesis, University of Texas at Dallas, 2005.

[56] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. In *Proceedings of the 2th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, volume 1995 of *LNCS*, pages 18–38. Springer, 2001.

[57] Pierre-Charles David and Thomas Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *Proceedings of the 5th International Symposium on Software Composition (SC)*, volume 4089 of *LNCS*, pages 82–97. Springer, 2006.

[58] Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of Computation Orchestration Via Timed Automata. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM)*, pages 226–245. Springer, 2006.

[59] Rémi Douence, Pascal Fradet, and Mario Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proceedings of the 1st ACM Conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188. Springer, 2002.

[60] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 141–150. ACM, 2004.

[61] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

[62] Roger Duke and Graeme Smith. Temporal Logic and Z Specifications. *Australian Computer Journal*, 21(2):62–66, 1989.

[63] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd workshop on Formal methods in software practice (FMSP)*, pages 7–15. ACM, 1998.

[64] EMF. Eclipse Modeling Framework. `http://www.eclipse.org/emf/`.

[65] Markus Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 68—-79, 1992.

[66] Ulfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, page 246. IEEE Computer Society, 2000.

[67] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 32–45. IEEE Computer Society, 1999.

[68] Ylies Falcone. You should Better Enforce than Verify. In *Tutorial of 1st International Conference on Runtime Verification (RV)*, 2010.

[69] David Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[70] Colin J. Fidge. Specification and Verification of Real-Time Behaviour Using Z and RTL. In *Proceedings of the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 393–409. Springer, 1991.

[71] Philip W. L. Fong. Access Control By Tracking Shallow Execution History. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 43–55. IEEE Computer Society, 2004.

[72] Howard Foster. *A Rigourous Approach to Engineering Web Service Compositions.* Phd thesis, Imperial College London, 2006.

[73] Xiao Fu, Xinyu Da, and Zhenhua Yu. Modeling Dynamic Software Architecture Based on π-Net. In *Proceedings of the 2nd International conference on Information and Communication Technologies (ICTTA)*, pages 2861–2865. IEEE, 2006.

[74] David Garlan. Software Architecture: a Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pages 91–101. ACM, 2000.

[75] David Garlan and Mary Shaw. An Introduction to Software Architecture. Technical Report CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, 1994.

[76] Dragan Gasevic and Devedzi Devedzic. Software Support for Teaching Petri Nets: P3. In *Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies (ICALT)*, pages 300–301. IEEE, 2003.

[77] GEF. Graphical Editing Framework. `http://www.eclipse.org/gef/`.

[78] Geri Georg, Indrakshi Ray, and Robert France. Using Aspects to Design a Secure System. In *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems*, pages 117–126. IEEE Computer Society, 2002.

[79] Virgil Gligor, Serban Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 172–183. IEEE, 1998.

[80] GMF. Graphical Modeling Framework. `www.eclipse.org/gmf`.

[81] Simon Godik and Tim Moses. eXtensible Access Control Markup Language (XACML). www.oasis-open.org/committees/xacml, 2003.

[82] Michael J. C. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1998.

[83] Iris Groher and Stefan Schulze. Generating Aspect Code from UML Models. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling (AOM)*, 2003.

[84] Nawal Guermouche, Olivier Perrin, and Christophe Ringeissen. Timed Specification For Web Services Compatibility Analysis. *Electronic Notes in Theoretical Computer Science*, 200(3):155–170, 2008.

[85] Elnar Hajiyev, Neil Ongkingco, Pavel Avgustinov, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Datalog as a Pointcut Language in Aspect-Oriented Programming. In *Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 667–668. ACM, 2006.

[86] Rachid Hamadi and Boualem Benatallah. A Petri Net-based Model for Web Service Composition. In *Proceedings of the 14th Australasian database conference*, pages 191–200, 2003.

[87] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability Classes for Enforcement Mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

[88] David Harel, Asaf Kleinbort, and Shahar Maoz. S2A: a Compiler for Multi-Modal UML Sequence Diagrams. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering (FASE)*, volume 4422 of *LNCS*, pages 121–124. Springer, 2007.

[89] Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[90] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using SAM. *Journal of Systems and Software*, 71(1-2):11–29, 2004.

[91] Hello2Morrow. SonarJ. http://www.hello2morrow.de/angebot/sonarj.php, 2007.

[92] Gerald H. Hilderink. Graphical modelling language for specifying concurrency based on CSP. *IEE Proceedings - Software*, 150(2):108–120, 2003.

[93] Charles Antony Richard Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(1):271–281, 1972.

[94] IBM Inc. Rational Rose Developer for Java. `http://www-01.ibm.com/software/awdtools/developer/rose/java/index.html`.

[95] Omando Inc. Omando. `http://www.ejb3.org/`.

[96] Asif Iqbal and Tzilla Elrad. Modeling Timing Constraints of Real-Time Systems as Crosscutting Concerns. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling (AOM)*, page 10, 2006.

[97] Xiaoping Jia and Sotiris Skevoulis. Code Synthesis Based on Object-Oriented Design Models and Formal Specifications. In *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 393–399. IEEE Computer Society, 1998.

[98] Diane Jordan and John Evdemon. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, 2007.

[99] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on Unified Modeling Language (UML)*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

[100] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Evaluation and Comparison of ADL Based Approaches for the Description of Dynamic of Software Architectures. In *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS)*, pages 189–195, 2005.

[101] Mohamed Hadj Kacem, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Describing dynamic software architectures using an extended UML model. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 1245–1249. ACM, 2006.

[102] Mohamed Hadj Kacem, Mohamed Nadhmi Miladi, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Towards a UML Profile for the Description of Dynamic Software Architectures. In *Proceedings of the International Conference on Component-Oriented Enterprise Applications (COEA)*, pages 25–39, 2005.

[103] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization Based Access Control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, page 120. IEEE Computer Society, 2003.

[104] Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini, and Mohamed Jmaiel. Specifying and Monitoring Temporal Properties in Web services Compositions. In *Proceedings of the 7th IEEE European Conference on Web Services (ECOWS)*, pages 148–157. IEEE Computer Society, 2009.

[105] Slim Kallel, Anis Charfi, and Mohamed Jmaiel. Using Aspects for Enforcing Formal Architecturals Invariants. *Electronic Notes in Theoretical Computer Science*, 215:5–21, 2008.

[106] Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. Combining Formal Methods and Aspects for Specifying and Enforcing Architectural Invariants. In *Proceedings of the 9th International Conference on Coordination Models and Languages (Coordination)*, volume 4467 of *LNCS*, pages 211–230. Springer, 2007.

[107] Slim Kallel, Anis Charfi, Mira Mezini, and Mohamed Jmaiel. Aspect-based Enforcement of Formal Delegation Policies. In *Proceedings of the 3th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 9–19. IEEE, 2008.

[108] Slim Kallel, Anis Charfi, Mira Mezini, Mohamed Jmaiel, and Karl Klose. From Formal Access Control Policies to Runtime Enforcement Aspects. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems (ESSoS)*, volume 5429 of *LNCS*, pages 16–31. Springer, 2009.

[109] Slim Kallel, Anis Charfi, Mira Mezini, Mohamed Jmaiel, and Andreas Sewe. A Holistic Approach for Access Control Policies: From Formal Specification to Aspect-based Enforcement. *International Journal of Information and Computer Security*, 3(3/4):337 – 354, December 2009.

[110] Slim Kallel, Mohamed Hajd Kacem, and Mohamed Jmaiel. Meidya. `www.redcad.org/projects/meidya`, 2010.

[111] Slim Kallel, Mohamed Hajd Kacem, and Mohamed Jmaiel. Modeling and enforcing invariants of dynamic software architectures. *Software and Systems Modeling*, page 23 pages, 2011. (to appear).

[112] Raman Kazhamiakin, Paritosh Pandya, and Marco Pistore. Representation, Verification, and Computation of Timed Properties in Web. In *Proceedings of the 4th IEEE International Conference on Web Services (ICWS)*, pages 497–504. IEEE, 2006.

[113] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[114] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[115] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. MaC: A Framework for Run-time Correctness Assurance of Real-Time Systems. Technical report, University of Pennsylvania, 1999.

[116] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[117] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case Studies for Method and Tool Evaluation. *IEEE Software*, 12(4):52–62, 1995.

[118] Karl Klose and Klaus Ostermann. Back to the Future: Pointcuts as Predicates over Traces. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FAOL)*, 2005.

[119] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A Graph-based Formalism for RBAC. *ACM Transactions on Information and System Security*, 5(3):332–365, 2002.

[120] Ivan Krechetov, Bedir Tekinerdogan, Alessandro Garcia, Christina Chavez, and Uirá Kulesza. Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design. In *Proceedings of the 8th International Workshop on Aspect-Oriented Modeling (AOM)*, 2006.

[121] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Bernard Coulette. Designing Dynamic Reconfiguration of Distributed Real Time Embedded Systems. In *Proceedings of the 10th annual international conference on New Technologies of Distributed Systems (NOTERE)*. IEEE Computer Society, 2010.

[122] Ingolf H. Krüger, Gunny Lee, and Michael Meisinger. Automating Software Architecture Exploration with M2Aspects. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools (SCESM)*, pages 51–58. ACM, 2006.

[123] Ingolf H. Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime Verification of Interactions: from MSCs to Aspects. In *Proceedings of the 7th international conference on Runtime verification (RV)*, pages 63–74. Springer, 2007.

[124] Ramnivas Laddad. *Aspect Oriented Refactoring*. Addison-Wesley Professional, 2006.

[125] Peter Bernard Ladkin. *The Logic of Time Representation*. PhD thesis, University of California, Berkeley, 1987. Chairman-Ralph Mckenzie.

[126] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[127] Leslie Lamport. TLZ (Abstract). In *Proceedings of the Z User's Workshop*, pages 267–268, Cambridge, UK, 1994.

[128] Yves Ledru. Identifying Pre-Conditions with the Z/EVES Theorem Prover. In *Proceedings of the 13th IEEE international conference on Automated software engineering (ASE)*, page 32. IEEE Computer Society, 1998.

[129] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[130] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing Non-safety Security Policies with Program Monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *LNCS*, pages 355–373. Springer, 2005.

[131] Xiaoli Liu and Rongqiang Fan. A Lightweight Framework for Code Generation from B Formal Specification. In *Proceedings of the 2nd International Workshop on Education Technology and Computer Science (ETCS)*, pages 133–136. IEEE, 2010.

[132] Sihem Loukil, Slim Kallel, Bechir Zalila, and Mohamed Jmaiel. Toward an Aspect Oriented ADL for Embedded Systems. In *Proceedings of the 4th European Conference on Software Architecture (ECSA)*, volume 6285 of *LNCS*, page 489–492. Springer, 2010.

[133] Imen Loulou, Ahmed Hadj Kacem, Mohamed Jmaiel, and Khalil Drira. Towards a Unified Graph-Based Framework for Dynamic Component-Based Architectures Description in Z. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS)*, pages 227–234. IEEE Computer Society, 2004.

[134] Mark Mahoney and Tzilla Elrad. Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines. In *Proceedings of the 6th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

[135] Sam Malek. Effective Realization of Software Architectural Styles with Aspects. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 313–316. IEEE Computer Society, 2008.

[136] Shahar Maoz and David Harel. From Multi-modal Scenarios to Code: Compiling LSCs into AspectJ. In *Proceedings of the 14th ACM International Symposium on Foundations of Software Engineering (FSE)*, pages 219–230. ACM, 2006.

[137] Shahar Maoz and David Harel. On Tracing Reactive Systems. *Software and Systems Modeling*, 2011. (To Appear).

[138] Keith McAlpine and Paul Golde. A new architecture for a collaborative authoring system. *Computer Supported Cooperative Work*, 2(3):159–174, 1994.

[139] Irwin Meisels and Mark Saaltink. The Z/EVES Reference Manual (for Version 1.5). Reference manual, ORA Canada, 1997.

[140] Patrick O'Neil Meredith, Dongyun Jin, Feng Chen, and Grigore Rosu. Efficient Monitoring of Parametric Context-Free Patterns. *Automated Software Engineering*, 17(2):149–180, 2010.

[141] Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[142] Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–99, 2003.

[143] Christie D. Michelsen, Wayne D. Dominick, and Joseph E. Urban. A Methodology for the Objective Evaluation of the user/system Interfaces of the MADAM System Using Software Engineering Principles. In *Proceedings of the 18th annual Southeast regional conference*, pages 103–109. ACM, 1980.

[144] Mohamed Nadhmi Miladi, Mohamed Jmaiel, and Mohamed Hadj Kacem. A UML Profile and a Fujaba Plugin for Modelling Dynamic Software Architectures. In *Proceedings of the Workshop on Model-Driven Software Evolution (MoDSE)*, pages 20–23. IEEE Computer Society, 2007.

[145] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM)*, volume 2999 of *LNCS*, pages 267–286. Springer, 2004.

[146] Till Mossakowski, Micheal Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and 4th International Conference on Temporal Logic (TIME-ICTL)*, pages 83–90. IEEE Computer Society, 2003.

[147] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[148] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM symposium on Principles of programming languages (POPL)*, pages 228–241. ACM, 1999.

[149] Luis Daniel Benavides Navarro, Mario Sudholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvee. Explicitly Distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 51–62. ACM, 2006.

[150] Gustaf Neumann and Mark Strembeck. An approach to engineer and enforce context constraints in an RBAC environment. In *Proceedings of the 8th ACM symposium on Access control models and technologies (SACMAT)*, pages 65–79. ACM, 2003.

[151] Angela Nicoara and Gustavo Alonso. Dynamic AOP with PROSE. In *Proceedings of the CAiSE Workshops (Vol. 2)*, pages 125–138. FEUP Edições, Porto, 2005.

[152] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[153] OMG. Catalog of UML Profile Specifications. `http://www.omg.org/technology/documents/profile\_catalog.htm`.

[154] OMG. UML Profile for Schedulability, Performance, and Time Specification. `http://www.omg.org/technology/documents/formal/schedulability.htm`, 2003.

[155] Flavio Oquendo. π-Method: a Model-driven Formal Method for Architecture-Centric Software Engineering. *SIGSOFT Software Engineering Notes*, 31(3):1–13, 2006.

[156] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[157] Lawrence C. Paulson. Isabelle: The Next Seven Hundred Theorem Provers. In *Proceedings of the 9th International Conference on Automated Deduction (CADE)*, volume 310 of *LNCS*, pages 772–773. Springer, 1988.

[158] Jaime Pavlich-Mariscal, Laurent Michel, and Steven Demurjian. A Formal Enforcement Framework for Role-Based Access Control Using Aspect-Oriented Programming. In *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 3713 of *LNCS*, pages 537–552. Springer, 2005.

[159] Jaime Pavlich-Mariscal, Laurent Michel, and Steven Demurjian. Enhancing UML to Model Custom Security Aspects. In *Proceedings of the 11th International Workshop on Aspect-Oriented Modeling (AOM)*, page 10, 2007.

[160] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, volume 2192 of *LNCS*, pages 1–24. Springer, 2001.

[161] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, volume 2192 of *LNCS*, pages 1–24. Springer, 2001.

[162] Stephan Philippi. Automatic Code Generation from High-Level Petri-Nets for Model Driven Systems Engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006.

[163] Mónica Pinto and Lidia Fuentes. AO-ADL: an ADL for Describing Aspect-Oriented Architectures. In *Proceedings of the 10th International Conference on Early Aspects*, volume 4765 of *LNCS*, pages 94–114. Springer, 2007.

[164] Franco Raimondi, James Skene, and Wolfgang Emmerich. Efficient online monitoring of web-service SLAs. In *Proceedings of the 16th ACM International Symposium on Foundations of Software Engineering (FSE)*, pages 170–180. ACM, 2008.

[165] Sowmiya Ramkarthik and Cui Zhang. Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z. In *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, pages 405–411. IEEE Computer Society, 2006.

[166] Indrakshi Ray, Robert B. France, Na Li, and Geri Georg. An Aspect-based Approach to Modeling Access Control Concerns. *Information & Software Technology*, 46(9):575–587, 2004.

[167] Amira Regayeg, Slim Kallel, Ahmed Hadj Kacem, and Mohamed Jmaiel. ForMAAD Method: An Experimental Design for Air Traffic Control. *International Transactions on Systems Science and Applications*, 1(4):327–334, 2006.

[168] Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture Modeling Language based on UML2.0. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 663–669. IEEE Computer Society, 2004.

[169] Krzysztof Sacha. Evaluation of Software Quality. In *Proceedings of the 2005 conference on Software Engineering: Evolution and Emerging Technologies*, pages 381–388. IOS Press, 2005.

[170] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access Control: Policies, Models, and Mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196. Springer, 2001.

[171] Andreas Schaad. Detecting Conflicts in a Role-Based Delegation Model. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, page 117. IEEE Computer Society, 2001.

[172] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A Model-Checking Approach to Analysing Organisational Controls in a Loan Origination Process. In *Proceedings of the 11th ACM symposium on Access control models and technologies (SACMAT)*, pages 139–149. ACM, 2006.

[173] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[174] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. A Role-Based Access Control Policy Verification Framework for Real-Time Systems. In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 13–20. IEEE Computer Society, 2005.

[175] Mary Shaw. Toward higher-level Abstractions for Software Systems. *Data & Knowledge Engineering*, 5(2):119–128, 1990.

[176] Mati Shomrat and Amiram Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–9. ACM, 2002.

[177] James Skene, Allan Skene, Jason Crampton, and Wolfgang Emmerich. The Monitorability of Service-Level Agreements for Application-Service Provision. In *Proceedings of the 6th international workshop on Software and performance*, pages 3–14. ACM, 2007.

[178] Ann E. Kelley sobel and Jim Alves-Foss. A Trace-Based Model of the chinese Wall Security Policy. In *Proceedings of the 1999 National Information System Security Conference*, 1999.

[179] Karsten Sohr, Michael Drouineaud, and Gail-Joon Ahn. Formal Specification of Role-based Security Policies for Clinical Information Systems. In *Proceedings of the 2005 ACM symposium on Applied computing (SAC)*, pages 332–339. ACM, 2005.

[180] Eunjee Song, Raghu Reddy, Robert France, Indrakshi Ray, Geri Georg, and Roger Alexander. Verifiable composition of access control and application features. In *Proceedings of the 10th ACM symposium on Access control models and technologies (SACMAT)*, pages 120–129. ACM, 2005.

[181] Mike Spivey. *The Z notation: a reference manual, Second Edition.* Prentice Hall International Ltd., 1992.

[182] International Standard. Information Technology — Z formal Specification Notation — Syntax, Type System and Semantics, 2002.

[183] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features. In *Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling (AOM)*, 2003.

[184] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Proceedings of the 5th Workshop on Runtime Verification (RV)*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 109–124. Elsevier, 2005.

[185] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive Scoping of Distributed Aspects. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 27–38. ACM, 2009.

[186] UPPAAL. `http://www.uppaal.com/`, 2009.

[187] Maarten H. van Emden and Robert A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the Assossiation for Computing Machinery*, 23(4):733–742, 1976.

[188] Tine Verhanneman, Frank Piessens, Bart De Win, Eddy Truyen, and Wouter Joosen. Implementing a modular access control service to support application-specific policies in CaesarJ. In *Proceedings of the 1st workshop on Aspect oriented middleware development*. ACM, 2005.

[189] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A Co-design Approach for Embedded System Modeling and Code Generation with UML and MARTE. In *Proceedings of the Design, Automation and Test in Europe Comference(DATE)*, pages 226–231. IEEE, 2009.

[190] Hua Wang, Yanchun Zhang, Jinli Cao, and Jian Yang. Specifying Role-Based Access Constraints with Object Constraint Language. In *Proceedings of the 6th Asia Pacific Web Conference (APWeb)*, volume 3007 of *LNCS*, pages 687–696. Springer, 2004.

[191] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23(9):8–23, 1990.

[192] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice Hall International Ltd., 1996.

[193] XPath. XML Path Language. `http://www.w3.org/TR/xpath`.

[194] Dianxiang Xu and Vivek Goel. An Aspect-Oriented Approach to Mobile Agent Access Control. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC)*, pages 668–673. IEEE Computer Society, 2005.

[195] Hong Yan, David Garlan, Bradley R. Schmerl, Jonathan Aldrich, and Rick Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 470–479. IEEE Computer Society, 2004.

[196] Zifu Yang and Tian Zhao. Improve Pointcut Definitions with Program Views. In *Proceedings of the 5th workshop on software engineering properties of languages and aspect technologies (SPLAT)*, page 9. ACM, 2007.

[197] Chunyang Yuan, Yeping He, Jianbo He, and Zhouyi Zhou. A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty. In *Proceedings of the 2nd SKLOIS Conference on Information Security and Cryptology (Inscrypt)*, volume 4318 of *LNCS*, pages 196–210. Springer, 2006.

[198] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A Rule-based Framework for Role based Delegation. In *Proceedings of the 6th ACM symposium on Access control models and technologies (SACMAT)*, pages 153–162. ACM, 2001.

[199] Xinwen Zhang, Sejong Oh, and Ravi Sandhu. PBDM: a flexible delegation model in RBAC. In *Proceedings of the 8th ACM symposium on Access control models and technologies (SACMAT)*, pages 149–157. ACM, 2003.

[200] Steffen Zschaler. Formal Specification of Non-Functional Properties of Component-based Software Systems. *Software and Systems Modeling*, 9(2):161–201, 2010.

Curriculum Vitae

- December 1, 1980
  Born in Sfax, Tunisia
- 09/1986 – 06/1992
  Primary School, 36 rue de 5 août, Sfax
- 09/1992 – 06/1999
  Secondary School Majida Boulila, Sfax
  Graduated with a high school diploma (Baccalauréat)
- 09/1999 – 06/2001
  Preparatory Institute for Engineers Study of Sfax
  University of Sfax, Tunisia
- 09/2001 – 06/2004
  National School of Engineers of Sfax
  University of Sfax, Tunisia
  Graduated with a computer science engineer diploma
- 09/2003 – 07/2005
  National School of Engineers of Sfax
  University of Sfax, Tunisia
  Graduated with a master degree
- 10/2006 – 07/2011
  PhD student in the Software Technology Group of Prof. Mira Mezini
  Darmstadt University of Technology, Germany
  Graduated with a doctoral degree